SHORTEST PATH PROBLEMS: DOMAIN RESTRICTION, ANYTIME PLANNING, AND MULTI-OBJECTIVE OPTIMIZATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Zachary Clawson

January 2017

SHORTEST PATH PROBLEMS: DOMAIN RESTRICTION, ANYTIME PLANNING, AND MULTI-OBJECTIVE OPTIMIZATION

Zachary Clawson, Ph.D.

Cornell University 2017

Optimal path problems arise in many applications and several efficient methods are widely used for solving them on the whole domain. However, practitioners are often only interested in the solution at one specific source point, i.e. the shortest path to the exit-set from a particular starting location. This thesis will focus on three separate, but related, problems of this form. We employ solution methods that discretize the computational domain and recover an approximate solution to the shortest path problem. These methods either solve the problem on a geometrically embedded graph or approximate the viscosity solution to a static Hamilton-Jacobi PDE.

Such paths can be viewed as characteristics of static Hamilton-Jacobi equations, so we restrict the computations to a neighborhood of the characteristic. We explain how heuristic under/over-estimate functions can be used to obtain a *causal* domain restriction, significantly decreasing the computational work without sacrificing convergence under mesh refinement. The discussed techniques are inspired by an alternative version of the classical A* algorithm on graphs. We illustrate the advantages of our approach on continuous isotropic examples in 2D and 3D. We compare its efficiency and accuracy to previous domain restriction techniques and analyze the behavior of errors under the grid refinement.

However, if the heuristic functions used are very inaccurate this can lead to A*-type methods providing little to no restriction. One solution is to scale-up the underestimate functions used so that they become more accurate on parts of the domain. However,

this will cause the algorithm to recover suboptimal, albeit locally optimal, solutions. These algorithms quickly produce an initial suboptimal solution that is iteratively improved. This ensures early availability of a good suboptimal path before the completion of the search for a globally optimal path. We illustrate the algorithm on examples where previous A*-FMM algorithms are unable to provide significant savings due to the poor quality of the heuristics.

Finally we present a related algorithm for finding optimal paths on graphs with respect to two criteria simultaneously. Our approach is based on augmenting the state space to keep track of the "budget" remaining to satisfy the constraints on secondary cost. The resulting augmented graph is acyclic and the primary cost can be then minimized by a simple upward sweep through budget levels. The efficiency and accuracy of our algorithm is tested on Probabilistic Roadmap graphs to minimize the distance of travel subject to a constraint on the overall threat exposure to enemy observers.

BIOGRAPHICAL SKETCH

Zachary Clawson was born in Charlotte, North Carolina on October 26, 1988 to Catherine Carrington Clawson and John David Clawson. At a young age he moved to the small coastal town of Manteo, North Carolina on Roanoke Island. Zach became interested in mathematics competitions throughout high school under the mentorship of his outstanding calculus teacher, Frank Vrablic. Zach attended NC State University from 2007-2011, graduating as a valedictorian. During the summer of 2010 he was accepted to an REU program at Cornell University to work with Professor Alexander Vladimirsky. The experience led him to attend Cornell in the Center for Applied Mathematics where he continued to work with Professor Vladimirsky on exciting and intriguing problems. For his first three years at Cornell he was supported under a National Science Foundation Graduate Research Fellowship.

To my mother, Catherine Carrington Clawson.

ACKNOWLEDGEMENTS

Thank you to all of my co-authors and colleagues who contributed to the work found in the main three chapters of this thesis. Chapter 2 was coauthored with Alexander Vladimirsky and Adam Chacon. Chapter 3 was written with many helpful discussions and advice from Alexander Vladimirsky. Chapter 4 was coauthored with Dennis Ding, Brendan Englot, Thomas A. Frewen, William M. Sisson, and Alexander Vladimirsky, and is a result of an internship with United Technologies Research Center. Thank you to all of my advisors and colleagues for a wonderful working experience, their time, and their support.

I would like to thank Professor Alexander Vladimirsky for his support throughout the years. It was through his calm guidance, helpful discussions, and general eagerness and excitement that I became involved and continued in this area of research. I am grateful to my other two committee members, John Guckenheimer and David Bindel, for their helpful input and guidance along the way.

I have to acknowledge the staff at United Technologies Research Center's autonomy program for an incredible internship working there during the summer of 2015.

I sincerely am grateful to all of the students in CAM that I have met throughout the years who continue to make our program a great and fostering environment. A special thanks to some of the CAM students in my year who were especially supportive: Isabel Kloumann, Danielle Toupo, Evan Randles, Ian Lizarraga, and Sumedh Joshi.

I have to thank my high school mathematics teacher, Frank Vrablic. I'm eternally grateful for the incredibly high level of education I received from him. I would not have studied mathematics at university had it not been for Frank Vrablic.

I sincerely thank my family for the support throughout the years. Thank you to Maple Clawson for so much nonacademic support during my time in graduate school.

This work was supported throughout the years by the National Science Foundation

Graduate Research Fellowship, the National Science Foundation Grant DMS-1016150, teaching assistantships in Cornell's Department of Mathematics, and the Center for Applied Mathematics.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Shortest path problems have numerous and surprising applications including optimal path-planning, image processing, shape-from-shading, wavefront propagation, financial trading, and many more [8, 68]. The numerical results and examples provided in this work are geared towards robotic path-planning applications in particular: how does a robot plan an optimal path from a source location $\boldsymbol{s}$ to a target location $\boldsymbol{t}$? We will focus on efficient algorithms for shortest path problems both on graphs and in continuous domains, the latter version leading to first-order non-linear PDEs in the planning space. The basic dichotomy is between Eulerian and Lagrangian approaches.

- *Eulerian* methods offer many advantages, namely the ability recover an approximate solution recovered that is guaranteed to converge to the true solution under refinement. Eulerian methods compute and store the solution at *all* discretized points in the domain, which can be useful in some applications. However, this can also be one of the largest drawbacks due to the computational cost and large memory requirement. For example, if a simple Cartesian grid is imposed on a hypercube $[0,1]^n \subset \mathbb{R}^n$ with m gridpoints per dimension the total number of gridpoints necessary is m^n. This poor scaling is known as the "curse of dimensionality".
- While *Lagrangian* methods are typically less computationally intensive in higher dimensions (they do not suffer from the curse of dimensionality), they suffer from several other drawbacks. In particular, they can be difficult to apply depending on the smoothness of the input functions and also may not recover the globally optimal trajectory. Lagrangian methods in the context of optimal path planning go under many names such as "Pontryagin's Maximum Principle" [63] or the "Characteristic Equations" [25] of an appropriate PDE (to be discussed). In contrast

to Eulerian methods, Lagrangian methods find the solution at a given location $\boldsymbol{s}$ in the domain $\Omega \subset \mathbb{R}^n$ (i.e. the optimal trajectory from that location). These methods are based on minimizing the cost functional subject to local variations in the trajectory, which leads to a two-point boundary value problems resulting in a coupled system of $2n$ ODEs.

Therefore the focus of this thesis will be on Eulerian-type algorithms that discretize the whole domain since they are guaranteed to recover the globally optimal path. There are two ways to compute the shortest path by discretizing the domain: either build a geometrically embedded graph and find the shortest path across the graph, or discretize the domain and solve a first-order non-linear PDE (the Hamilton-Jacobi-Bellman PDE) whose solution at meshpoints corresponds to the amount of time it takes to exit the domain (reach the target location $\boldsymbol{t}$). The theories used in developing algorithms for both settings are highly interconnected and there are still an abundance of open questions concerning their relationship.

Chapters 2 and 3 introduce ideas from graph theory to solve these HJB PDE in an efficient manner at a specific location in the domain. These two chapters aim to address the "curse of dimensionality" by bringing some of the advantages of Lagrangian methods into the Eulerian approach. In Chapter 4 we develop new algorithms for solving bi-objective shortest path problems on geometrically embedded graphs that are built using the Probabilistic RoadMap planner [40].

1.1 Shortest paths on graphs

There are many algorithms for solving shortest path problems on graphs, each with its own computational complexity and advantages / disadvantages. Detailed introductions

are provided in §2.2, §3.2, and §4.2.1, however we will briefly overview the setting and algorithms here.

1.1.1 Problem Statement

A graph $\mathcal{G}$ is defined by a set of vertices $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_M, \boldsymbol{x}_{M+1}\}$ and edges E. A transition time penalty is assigned to each edge and is given by $C_{ij} = C(\boldsymbol{x}_i, \boldsymbol{x}_j) > 0$ with the convention that $C_{ij} = +\infty$ if there is no edge from vertex $\boldsymbol{x}_i$ to $\boldsymbol{x}_j$. For simplicity here assume that the graph is undirected and the neighbors of a node $\boldsymbol{x}_i$ are given by $N(\boldsymbol{x}_i) = \left\{\boldsymbol{x}_j \mid C_{ij} < +\infty\right\}$ and that $|N(\boldsymbol{x}_i)| = \kappa \ll M$. The goal is to recover the value function $U : X \to [0, +\infty]$ given by

$$U(\boldsymbol{x}_i) \;=\; \text{the minimum total time to travel from } \boldsymbol{x}_i \text{ to } \boldsymbol{t} = \boldsymbol{x}_{M+1}$$

where $\boldsymbol{t} = \boldsymbol{x}_{M+1}$ is the specified "target" node (exit-set) in the graph, i.e. the location we want to travel to. Thus WLOG we can assume that $U(\boldsymbol{t}) = 0$.

Bellman's Optimality Principle [6] provides the dynamic programming equation

$$U(\boldsymbol{x}_i) \;=\; \min_{\boldsymbol{x}_j \in N(\boldsymbol{x}_i)} \left\{U(\boldsymbol{x}_j) + C(\boldsymbol{x}_i, \boldsymbol{x}_j)\right\} \qquad \forall\, i = 1, 2, \ldots, M. \tag{1.1}$$

Solving this system of coupled non-linear equations at every node in the graph allows for the recovery of the optimal path from every location. The most straight-forward approach is known as *value iteration* where an initial overestimate guess is provided for U and (1.1) is applied iteratively until U converges to the correct solution. However, much more efficient algorithms can be built by using the properties of U along with basic data structures. We overview these approaches in the next subsection.

1.1.2 Algorithms

There are two general classes of algorithms used to solve for U at every location:

- **Label-correcting methods.** Label-correcting algorithms maintain a list of "open" (or "considered") nodes that have been recently updated.
  - On each iteration a node $\boldsymbol{x}_i$ is popped from the *open* list.
  - Next, for each neighbor $\boldsymbol{x}_j \in N(\boldsymbol{x}_i)$ a tentative value of $\tilde{U}_j$ is computed, and if $\tilde{U}_j < U_j$ then $\boldsymbol{x}_j$ is placed on the open list and $U_j \leftarrow \tilde{U}_j$.
  - The process continues until the *open* list is empty.

  A basic version is known as the Bellman-Ford algorithm [27] and there are many variations of it such as "Large Labels Last" [10], "Small Labels First" [9], "D'Esopo-Pape" [56], etc. See [16] for a comprehensive review.
- **Label-setting methods.** These methods guarantee that every node will be updated at most κ (the maximum number of neighbors) times before the algorithm converges. In particular, once a node becomes permanently 'labeled' (i.e. 'closed' or 'accepted') that node will never be updated again. Since the problems of focus in this thesis consist of finding the solution at one specific source node $\boldsymbol{s}$, label-setting algorithms are preferred as computations can be terminated immediately once $\boldsymbol{s}$ becomes permanently labeled. The classical label-setting algorithm on graphs is Dijkstra's algorithm [22], however many modifications have been proposed such as SA* [32], bidirectional Dijkstra's algorithm [61], Dial's algorithm [21], and more. See [8, 87] for general overviews.

Dijkstra's algorithm operates similarly to the generic label-correcting method described above with one modification to the first step: *On each iteration the 'considered'*

node $\boldsymbol{x}_i$ with smallest value $U(\boldsymbol{x}_i)$ is popped from the considered (open) list. This modification guarantees that once a node is popped from the 'considered' list, it is 'accepted' and will never re-enter. However, this requires an additional data structure such as a min-heap to efficiently remove the node with smallest value from the 'considered' nodes. Thus the computational complexity is $O(M \log M)$ where the log-term comes from the need to maintain the heap.

1.2 Continuous shortest path problem

One major drawback to discretizing the domain via an embedded graph is that the paths are often 'rigid' and not truly 'continuous domain optimal' since it would require extremely high refinement and connectivity to come close to approximating a smooth curve.[1] For instance, consider a grid-like graph in $\mathbb{R}^2$ with grid-spacing h as shown in Figure 1.1A. In Figure 1.1A, any path from $\boldsymbol{s}$ to $\boldsymbol{t}$ that travels strictly south or west at each transition is a shortest path with length $7h$. Further refinement of the grid-spacing will not change the length of the shortest path(s) as illustrated in Figure 1.1B. Increasing the connectivity of the graph will result in a reduction of error, but does not necessarily eliminate the error in the limit.

[1] Note that the convergence of graph-based sampling methods such as the Probabilistic RoadMap algorithm and the Rapidly-exploring Random Tree algorithm is addressed in [40].

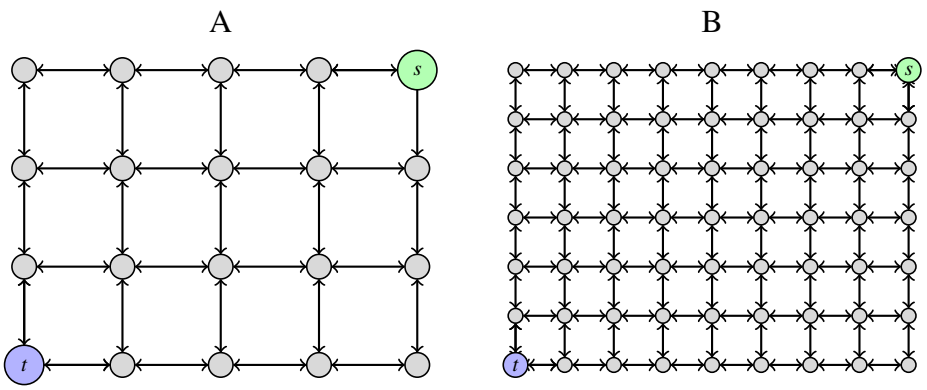


Figure 1.1: Examples of grid-like graphs.

In this situation graph-based algorithms would suffer from two main problems: (1) the obvious curse of dimensionality that comes along with dynamic programming; and (2) the lack of convergence to the solution of the optimal control problem [81] unless special precautions are taken in regards to the rate of refinement of the grid with respect to the neighborhood connectivity of the nodes [38].

As a result, alternative 'continuous' approaches are necessary to find an approximate solution that will converge to the true solution under grid refinement. This section will outline the derivation of the static Hamilton-Jacobi-Bellman PDE, particularly the Eikonal PDE that is used for isotropic optimal control problems.

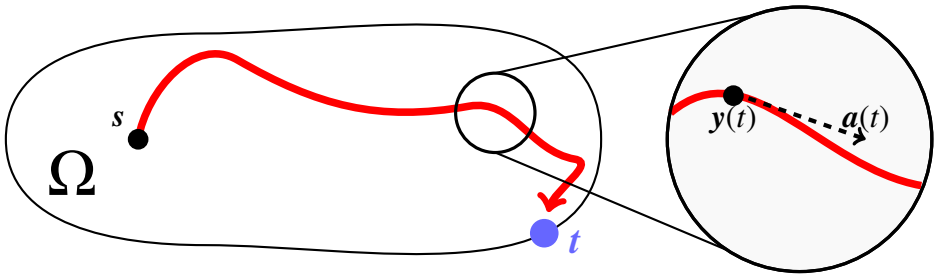


Figure 1.2: Example of an optimal path $\boldsymbol{y}(\cdot)$ in the domain $\Omega \subset \mathbb{R}^n$ from $\boldsymbol{s}$ to $\boldsymbol{t}$. At each time t the control $\boldsymbol{a}(t) \in S^{n-1}$ determines the direction of motion.

1.2.1 Problem Statement

Derivation of Eikonal PDE

Define the trajectory of motion by $\boldsymbol{y}(t)$ for $t \in [0, T]$ where

$$\begin{cases} \boldsymbol{y}'(t) &= f(\boldsymbol{y}(t))\boldsymbol{a}(t) & \text{in } \Omega \\ \boldsymbol{y}(0) &= \boldsymbol{x} & \text{for a given } \boldsymbol{x} \in \Omega. \end{cases} \tag{1.2}$$

Notationally $\boldsymbol{y}(t)$ is the trajectory in Ω starting at the point $\boldsymbol{x}$, the control function is $\boldsymbol{a} : \mathbb{R} \to S^{n-1} \subset \mathbb{R}^n$, and $f : \mathbb{R}^n \to [0, \infty)$ is the speed of motion at every location.

The process terminates once $\boldsymbol{y}(t)$ enters some exit-set $\mathcal{Q} \subset \partial\Omega$ for the first time $t \geq 0$, at which time an exit-time penalty is incurred given by the function $q : \mathcal{Q} \to \mathbb{R}$. For simplicity in Chapters 2 and 3 (where this approach is used) we will assume that $\mathcal{Q}$ is given by a single point, i.e. $\mathcal{Q} = \{\boldsymbol{t}\}$, and the exit-penalty is zero, i.e. $q \equiv 0$. The goal is to minimize the final exit time starting at $\boldsymbol{x}$ given by

$$T(\boldsymbol{x}, \boldsymbol{a}(\cdot)) \;=\; \min\{t \geq 0 \mid \boldsymbol{y}(t) \in \mathcal{Q}\},$$

and we define the *value function*

$$u(\boldsymbol{x}) \;=\; \inf_{|\boldsymbol{a}(\cdot)|=1} \{T(\boldsymbol{x}, \boldsymbol{a}(\cdot))\},$$

which can be interpreted as a continuous equivalent of (1.1). We will now present a formal derivation of the Eikonal PDE that u must satisfy if it is smooth.

By the Bellman's Optimality Principle [6]

$$u(\boldsymbol{x}) \;=\; \inf_{|\boldsymbol{a}(\cdot)|=1} \{\tau + u(\boldsymbol{y}(\tau))\}.$$

Performing a Taylor expansion on $u(\boldsymbol{y}(\tau))$ about $\tau = 0$ yields

$$\begin{aligned} u(\boldsymbol{x}) &= \inf_{|\boldsymbol{a}(\cdot)|=1} \left\{ \tau + u(\boldsymbol{y}(0)) + \frac{d}{ds}\left[u(\boldsymbol{y}(s))\right]_{s=0} \tau^1 + O(\tau^2) \right\} \\ &= \inf_{|\boldsymbol{a}(\cdot)|=1} \left\{ \tau + u(\boldsymbol{x}) + \tau f(\boldsymbol{x}) \nabla u(\boldsymbol{x}) \cdot \boldsymbol{a}(0) + O(\tau^2) \right\}. \end{aligned}$$

Subtracting $u(\boldsymbol{x})$ from both sides, dividing by τ, and allowing for $\tau \downarrow 0$ we arrive at the Hamilton-Jacobi-Bellman equation given by

$$\min_{|\boldsymbol{a}|=1} \{\nabla u(\boldsymbol{x}) \cdot \boldsymbol{a} f(\boldsymbol{x}) + 1\} \;=\; 0 \qquad \boldsymbol{x} \in \Omega \tag{1.3}$$

with boundary condition $u(\boldsymbol{t}) = 0$.

We may simplify (1.3) by realizing the optimal control value (the argmin of (1.3)) is $\boldsymbol{a} = -\nabla u(\boldsymbol{x}) / |\nabla u(\boldsymbol{x})|$, yielding the well-known Eikonal equation:

$$\begin{cases} |\nabla u(\boldsymbol{x})| &= 1/f(\boldsymbol{x}) \qquad \forall\, \boldsymbol{x} \in \Omega \\ u(\boldsymbol{t}) &= 0 \end{cases} \tag{1.4}$$

Viscosity Solutions

This derivation works as long as $\nabla u(\boldsymbol{x})$ is defined. However, it is very easy to see that $\nabla u(\boldsymbol{x})$ may not be defined for all $\boldsymbol{x} \in \Omega$. For example, consider the simple case when

$\Omega = [0,1] \subset \mathbb{R}^1$ with $\mathcal{Q} = \{0,1\}$ and $f(x) \equiv 1$ yielding the equation

$$\begin{cases} |u'| &= 1 \quad \text{on } (0,1) \\ u(0) &= u(1) = 0. \end{cases} \tag{1.5}$$

In this case, the value function for the optimal control problem is shown in Figure 1.3A and is explicitly given by $u(x) = 0.5 - |x - 0.5|$ whose derivative is undefined at $x = 0.5$. There are actually infinitely many weak solutions that satisfy this differential equation almost everywhere. For example, the curve shown in Figure 1.3B satisfies (1.5) almost everywhere, but does not correspond to the solution of the optimal control problem.

Remark 1. *When $\nabla u(\boldsymbol{x})$ does not exist at a point $\boldsymbol{x}$ this location is known as a "shock-line" in the value function $u(\boldsymbol{x})$. Locations where there are discontinuities in ∇u indicates points starting from which there is more than one optimal trajectory. In the simple $\mathbb{R}^1$ example stated above there are two optimal trajectories from the point $x = 0.5$: either move directly to the left or to the right since both are trajectories that minimize the time-to-exit. See §2.4.3 and Figure 2.15 for an example where we illustrate this phenomenon in $\mathbb{R}^2$.*

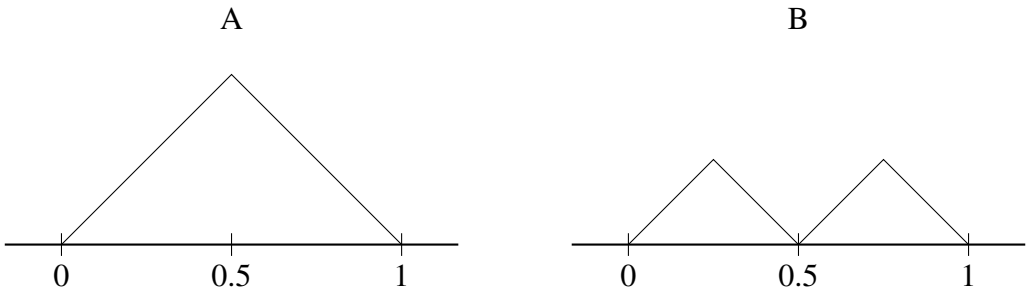


Figure 1.3: Two weak solutions to a simple one-dimensional version of the Eikonal equation given in (1.5). The solution on the left is the unique viscosity solution corresponding to the value function of the optimal control problem. The solution on the right is a weak solution but does not satisfy the conditions of a viscosity solution.

The need to select 'the correct' weak solution led to the definition of *viscosity solu-*

tions, a theory developed in the 1980s by Crandall and Lions [19]. A viscosity solution of (1.4) is any continuous function $u(\boldsymbol{x})$ that satisfies the following conditions [4]:

1. For any $\varphi \in C^1$, if $\boldsymbol{x}$ is a local maximum for $u - \varphi$, then $|\nabla\varphi(\boldsymbol{x})|\, f(\boldsymbol{x}) \leq 1$.
2. For any $\varphi \in C^1$, if $\boldsymbol{x}$ is a local minimum for $u - \varphi$, then $|\nabla\varphi(\boldsymbol{x})|\, f(\boldsymbol{x}) \geq 1$.

In fact, since φ is a C^1 test function we can assume that $u(\boldsymbol{x}) = \varphi(\boldsymbol{x})$ at the location where the maximum / minimum occurs by adding a constant. If u is smooth at $\boldsymbol{x}$, then $\nabla u = \nabla\phi$, implying that (1.4) holds at $\boldsymbol{x}$ in the classical sense. The uniqueness of viscosity solutions can be derived by assuming Lipschitz continuity of the Hamiltonian in $\boldsymbol{x}$ and $\nabla u(\boldsymbol{x})$ PDE [25]. For the Eikonal PDE, that translates to requiring Lipschitz continuity of $f(\boldsymbol{x})$. The proof of existence is constructive: Bellman's Optimality Principle can be used to show that the value function of the control problem is in fact a viscosity solution.

Returning to the simple one-dimensional example presented above we can now clearly see that $u(x) = 0.5 - |x - 0.5|$ shown in Figure 1.3A is the only one (out of infinitely many Lipschitz continuous weak solutions) that satisfies these requirements. For instance, suppose $u(x)$ satisfies (1.3) almost everywhere, but has a local minimum at $x = 0.5$ as illustrated in Figure 1.3B. For this particular weak solution, we can choose the test function $\varphi(x) = -(x - 0.5)^2$ that produces a local minimum for $u - \varphi$ at $x = 0.5$. However, we have that $|\varphi'(0.5)| = 0 \not\geq 1$. Therefore this weak solution, and any other that obtains a local minimum on $(0, 1)$, cannot be a viscosity solution.

1.2.2 Algorithms and Implementation Details

The Fast Marching Method (FMM) [68, 81] is the generalization of Dijkstra's algorithm with the slight modification that the update formula at each gridpoint changes from (1.1)

to a more complicated expression. The update formula for a gridpoint now relies on n of its neighbors in $\mathbb{R}^n$ and is computationally more expensive than the update formula on graphs (that uses only one neighboring node at a time to produce a new potential value).

For example, in $\mathbb{R}^2$ approximating u in (1.4) by a function U and using upwind finite differences on a uniform grid with spacing h leads to the equation

$$\left(\frac{U_{ij} - U_H}{h}\right)^2 + \left(\frac{U_{ij} - U_V}{h}\right)^2 = \frac{1}{f^2(\boldsymbol{x})}, \tag{1.6}$$

where $U_{ij} = U(\boldsymbol{x}_{ij})$ is the value to compute at the gridpoint $\boldsymbol{x}_{ij}$, $U_H = \min\left\{U_{i-1,j}, U_{i+1,j}\right\}$, and $U_V = \min\left\{U_{i,j-1}, U_{i,j+1}\right\}$ (with the convention that if $\boldsymbol{x}_{ij} \notin \Omega \implies U_{ij} = +\infty$). The values U_H and U_V represent the minimum values in the horizontal and vertical directions, i.e. the "upwinding direction". This is because optimal trajectory should travel in the direction of smaller U-values – similar to the one-dimensional example presented in the previous §1.2.1.

The increase in computational complexity comes from solving the quadratic equation in (1.6), which involves a more expensive square-root computation each time a gridpoint needs an update. The produced solution must satisfy the *upwinding condition* that $U_{ij} > \max\{U_H, U_V\}$. If neither root satisfies this condition, then a lower-dimensional update must be performed – in the case of $\mathbb{R}^2$ a one-dimensional update of $U_{ij} = \min\{U_H, U_V\} + h/f(\boldsymbol{x}_{ij})$ would be performed. See §2.3, §3.3, and [16] for more details.

After the approximate solution $U(\boldsymbol{x})$ is obtained on the grid or mesh, the optimal path from any location can be obtained via gradient descent. For all numerical results in this paper, we programmed a simple steepest descent library that searched for the next point to move to by considering a small S^{n-1} neighborhood about the current location and finely sampling points in this neighborhood. We were not concerned with optimizing this library since we considered this an inexpensive post-processing step. If descent

is initiated from a point $\boldsymbol{x}_0$ where $\nabla u(\boldsymbol{x}_0)$ exists, then $\nabla u(\boldsymbol{x})$ will exist for all points $\boldsymbol{x}$ encountered by the steepest descent. However, if $\nabla u(\boldsymbol{x}_0)$ does not exist then the initial direction of motion must be arbitrarily chosen or chosen by the user, and then for any $\boldsymbol{x} \neq \boldsymbol{x}_0$ encountered $\nabla u(\boldsymbol{x})$ will exist.

We will consider examples in $\mathbb{R}^2$ and $\mathbb{R}^3$ in this dissertation, but the generalization of our algorithms to higher dimensions is straightforward. However, in much higher dimensions the curse of dimensionality will begin to hinder performance. One idea to reduce the memory requirement is to allocate the gridpoints on-the-fly as they become needed. This idea was presented and implemented on graphs in [46], however we chose not to implement it for simplicity and since our examples are in $\mathbb{R}^2$ and $\mathbb{R}^3$ where memory requirements are less intensive.

1.3 Domain Restriction Techniques

In Chapter 2 we discuss the single-source/single-target optimal trajectory problem and apply ideas from A* algorithms on graphs to restrict the computations to a neighborhood of the optimal trajectory. The A* techniques presented here can be viewed as an attempt to bring some of the advantages of Lagrangian methods into the Eulerian approach of the Fast Marching Method. These restriction techniques modify the 'considered' heap used in Fast Marching Method by using a heuristic estimate to help center computations along the optimal path. For these continuous problems, we compare the classical A* algorithm ("Standard A*") [32] to a slightly different version of the A* algorithm ("Alternative A*") found in Bertsekas' 'Dynamic Programming and Optimal Control' textbook [8]. The classical SA* approach modifies the order in which nodes are popped from the 'considered' heap and labeled as 'accepted'. The AA* approach works differently by

not allowing nodes to become a part of the 'considered' heap if the estimates determine that the node is too far from the optimal path.

Several previous works applied the SA* algorithm to continuous problems discretized on a grid by modifying the Fast Marching Method. However, these publications either did not acknowledge the existence of error [26] (due to heuristically scaling down the underestimate used) or acknowledged additional error produced by the algorithm without any further rigorous investigation [57, 58, 59, 55]. As Yershov & LaValle first pointed out (see [84, 85] and §2.3.3), the type of the mesh used has an impact on the *consistency condition* necessary for Standard A* to produce zero additional error.

We show that on Cartesian grids the SA*-FMM approach does not produce a solution that converges to a viscosity solution under grid refinement. In contrast, our new modification of the AA*-FMM algorithm does indeed converge with an inconsistent (but admissible) heuristic function.

1.4 Anytime algorithms: on-demand path-planning

In Chapter 3 we study problems where the A* algorithms presented in Chapter 2 fail to provide sufficient restriction. This typically occurs when the heuristic underestimate function φ used by the A* algorithms is highly inaccurate (usually due to large variations of the speed within the domain). One remedy to this used on graphs is called "Weighted A*" where the heuristic is scaled by a constant $w \geq 1$ and the solution is computed using SA* with an inconsistent heuristic $w\varphi$.

Remark 2. *On graphs, if an inconsistent heuristic of this form is used then the sub-optimality of the produced solution can be bounded:* $\tilde{U}(\boldsymbol{s}) \leq wU(\boldsymbol{s})$ *where* $\tilde{U}$ *is the WA* solution and* U *is the true solution (e.g. computed via Dijkstra's algorithm) [80]. It*

is future work to prove such a sub-optimality bound for path-planning with continuous problems.

This can be viewed as an overoptimistic version of the domain restriction techniques performed in Chapter 2. WA* can be useful for single search queries to produce a viable path, however it can be difficult to determine a value of w a priori that produces a solution with desired restriction and sub-optimality. Thus one reasonable strategy is to perform repeated WA* search with different values of w. However, simply performing repeated WA* searches can lead to wasteful computations and this observation led to the invention of Anytime A* algorithms on graphs.

On graphs, the first Anytime A* algorithm [46] operates by iteratively performing the Weighted A* algorithm, starting out with a large weight w and decreasing it until $w\varphi$ is a consistent heuristic. The algorithm makes use of additional data structures to store information from previous iterations in order to reduce the number of computations necessary. We have modified two different Anytime A* algorithms to work for continuous problems and numerically study the convergence of the solution over time: Anytime Repairing A* [46] (described above) and another version called Anytime Non-parametric A* [7].

1.5 Multi-Objective Optimization

In Chapter 4 we introduce a new algorithm for bi-objective optimization on graphs. Similar to the approach of Chapter 3, there is the component of finding multiple paths subject to some additional constraint(s). However, in this situation these constraints are not based on computational time (like in the Anytime algorithms), but rather based on physical constraints posed on the paths themselves.

The application considered in this chapter is planning the shortest path (minimize the primary cost, e.g. distance) subject to constraints on the accumulated secondary cost (e.g. exposure to an enemy observer). First, a graph is built over the physical space using the Probabilistic RoadMap planner [40]. We then augment the state space to keep track of the remaining 'budget' for the secondary cost. Effectively this takes a copy of the graph generated to represent the physical space and makes a copy for every "allowable" budget level. If the secondary costs are strictly positive, this leads to an algorithm where an upward-sweep may be performed at a computational cost of $O(mn)$ where n is the number of nodes in the original graph and m is the number of budget levels that have been discretized. However, if the secondary costs are allowed to be zero then Dijkstra's algorithm must be applied, leading to a slightly higher computational cost of $O(mn\log(mn))$.

The main objective of this section is to illustrate that our new algorithm can very quickly approximate what is known as the "Pareto Front". The Pareto Front in the objective space consists of pairs of primary and secondary costs (P,S) that correspond to a given path such that no other path can decrease one of the objective costs without increasing the other. We are interested in studying the convergence of the Pareto Front as the number of nodes in the graph grows or under refinement of the number of budget levels. The results show that our algorithm is not only fast and accurate, but recovers an approximation to the *entire* Pareto Front. A more common/competing approach based on the "scalarization algorithm" (see [51] and §4.4.4) is unable to recover the non-convex portions of the Pareto Front.

We illustrate the algorithm with numerical examples on a two-dimensional state space in §4.4 and well as present a field test done on a real robotics experiment in §4.5.

CHAPTER 2

CAUSAL DOMAIN RESTRICTION FOR EIKONAL EQUATIONS

2.1 Introduction

The Eikonal equation

$$\begin{cases} |\nabla u(\boldsymbol{x})|\, f(\boldsymbol{x}) = 1 & \forall\ \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}) = q(\boldsymbol{x}) & \forall\ \boldsymbol{x} \in Q \subseteq \partial\Omega, \end{cases} \tag{2.1}$$

arises naturally in many applications including continuous optimal path planning, computational geometry, photolithography, optics, shape from shading, and image processing [68]. One natural interpretation for the solution of (3.1) comes from isotropic time-optimal control problems. For a vehicle traveling through $\bar{\Omega}$, f describes the speed of travel and q gives the exit time-penalty charged on Q. In this framework, $u(\boldsymbol{x})$ is the *value function*; i.e., the minimum time to exit $\bar{\Omega}$ through Q if we start from a point $\boldsymbol{x} \in \Omega$. The characteristic curves of the PDE (3.1) define the optimal trajectories for the vehicle motion.

The value function is Lipschitz continuous, but generally is not smooth on Ω. (The gradient of u is undefined at all points for which an optimal trajectory is not unique.) Correspondingly, the PDE (3.1) typically does not have a smooth solution and admits infinitely many Lipschitz continuous weak solutions. Additional conditions introduced in [19] are used to restore the uniqueness: the *viscosity solution* is unique and coincides with the value function of the above control problem.

In the last 20 years, many fast numerical methods have been developed to solve

This chapter is based on the paper *Causal Domain Restriction for Eikonal Equations* by Z. Clawson, A. Chacon, & A. Vladimirsky, SIAM Journal on Scientific Computing, Volume 36, Issue 5, pp. A2478-A2505, 2014. This publication is listed as [18] in the bibliography.

(3.1) *on the entire domain* $\bar{\Omega}$; e.g., see [17, 67, 81, 88]. Many of these fast methods were inspired by classical label-correcting and label-setting algorithms on graphs; e.g., Sethian's Fast Marching Method [67] mirrors the logic of the classical Dijkstra's algorithm [22], which finds the minimum time to a target-node from every other node in the graph.

Our focus here is on a somewhat different situation, with the solution needed *for one specific starting position only*. On graphs, an A* modification of Dijkstra's method [32] is widely used for similar *single source / single target* shortest path problems.

There have been several prior attempts to extend A* techniques to algorithms for continuous optimal trajectory problems, but all of them have significant drawbacks: these methods either produce additional errors that do not vanish under numerical grid refinement [55, 57, 58, 59], or provide much more limited computational savings [26, 84, 85]. We believe that these disadvantages stem from an overly faithful mirroring of the "standard" A* on graphs. Our own approach is based on an alternative version of the A* algorithm [8] that has clear advantages in continuous optimal control problems. Numerical testing confirms that our method is both efficient (in terms of the percentage of domain restriction) and convergent under grid refinement.

We begin by reviewing two flavors of A* techniques on graphs in §2.2. We then describe the standard Fast Marching Method and its various A*-type modifications in §2.3. The numerical tests in §2.4 are used to compare the efficiency and accuracy of competing domain restriction techniques. §2.5 contains convergence analysis of the alternative A* under grid refinement, exploiting the probabilistic interpretation of the discretized equations. We discuss the limitations of our approach and directions of future work in §2.6.

2.2 Domain Restriction Techniques on Graphs

We start by defining the shortest path problem on a graph:

- A graph $\mathcal{G}$ is defined by a set of nodes (vertices) $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{M+1} = \boldsymbol{t}\}$ and a set of directed arcs between these nodes.
- Along each arc we prescribe a transition time penalty $C(\boldsymbol{x}_i, \boldsymbol{x}_j) = C_{ij} > 0$, and assume $C_{ij} = +\infty$ if there is no transition from $\boldsymbol{x}_i$ to $\boldsymbol{x}_j$.
- The sets of *in-neighbors* and *out-neighbors* of a node $\boldsymbol{x}_j$ are respectively defined by

$$N_j^- \;=\; N^-(\boldsymbol{x}_j) \;\triangleq\; \left\{\boldsymbol{x}_i \mid C_{ij} < +\infty\right\}, \qquad N_j^+ \;=\; N^+(\boldsymbol{x}_j) \;\triangleq\; \left\{\boldsymbol{x}_k \mid C_{jk} < +\infty\right\}.$$

- Assume the graph is *sparsely connected*, i.e. $|N^{\pm}(\boldsymbol{x}_i)| \leq \kappa \ll M \;\forall\; \boldsymbol{x}_i \in X$ for some fixed $\kappa \in \mathbb{N}$.
- The **goal** is to find the *"value function"* $U : X \to [0, +\infty)$, defined as

$$U_i \;=\; U(\boldsymbol{x}_i) \;\triangleq\; \text{the minimum total time to travel from } \boldsymbol{x}_i \text{ to } \boldsymbol{t} = \boldsymbol{x}_{M+1}.$$

Naturally, $U_{M+1} = 0$. On the rest of the graph, *Bellman's Optimality Principle* [6] yields a coupled system of M nonlinear equations:

$$U_i \;=\; \min_{\boldsymbol{x}_j \in N_i^+} \left\{U_j \;+\; C_{ij}\right\}, \qquad \forall\; i = 1, 2, \ldots, M. \tag{2.2}$$

Once the value function is known, an optimal path from any node to the target $\boldsymbol{x}_{M+1}$ can be quickly recovered by recursively transitioning to the minimizing neighbor. A straight-forward iterative method for solving the system (2.2) would result in $O(M^2)$ computational cost. Fortunately, this system is *monotone causal*: U_i cannot depend on U_j unless $U_i > U_j$. This observation is the basis of the classical Dijkstra's method,

which recovers the value function on the entire graph in $O(M \log M)$ operations [22]. In Dijkstra's method, all nodes are split into three classes: FAR (no value yet assigned), CONSIDERED (assigned a tentative value), or ACCEPTED (assigned a permanent value).

Algorithm 1: *Dijkstra's Algorithm*

Initialization:

1 $U_i \leftarrow +\infty$ and mark $\boldsymbol{x}_i$ as FAR for $i = 1, 2, \ldots, M$
2 $U(\boldsymbol{t}) \leftarrow 0$ and mark $\boldsymbol{t}$ as ACCEPTED.
3 For all $\boldsymbol{x}_i \in N^-(\boldsymbol{t})$, mark as CONSIDERED and $U_i \leftarrow C(\boldsymbol{x}_i, \boldsymbol{t})$

Algorithm :

4 **while** $\exists$ *a* CONSIDERED *node* **do**
5 Find the CONSIDERED node $\boldsymbol{x}_j$ with minimal U-value and mark as ACCEPTED
6 **for** $\boldsymbol{x}_i \in N_j^-$ *such that* $U_i > U_j$ *and* $\boldsymbol{x}_i$ *is* FAR *or* CONSIDERED **do**
7 $\tilde{U} \leftarrow U_j + C_{ij}$
8 **if** $\tilde{U} < U_i$ **then**
9 $U_i \leftarrow \tilde{U}$
10 Mark $\boldsymbol{x}_i$ as CONSIDERED

Figure 2.1: Dijkstra's algorithm.

Efficient implementations usually maintain the CONSIDERED nodes as a binary heap, resulting in the $\log M$ term in the computational complexity.

2.2.1 Estimates for "single-source / single-target" problems.

If we are only interested in an optimal path from a single starting location $\boldsymbol{s} \in X$, Dijkstra's method can be terminated as soon $\boldsymbol{s}$ becomes ACCEPTED. (This changes the stopping criterion on line 4 of the pseudocode.) Other modifications of the algorithm can be introduced to further reduce the computational cost on this narrower problem. Consider a function

$$V_i \;=\; V(\boldsymbol{x}_i) \;\triangleq\; \text{minimum total time to travel from } \boldsymbol{s} \text{ to } \boldsymbol{x}_i.$$

Any node $\boldsymbol{x}_i$ lying on an optimal path from $\boldsymbol{s}$ to $\boldsymbol{t}$ must satisfy $U_i + V_i = U(\boldsymbol{s}) = V(\boldsymbol{t})$. This provides an obvious relevance criterion, since for any $\boldsymbol{x}_i$ that is not on an optimal path, $U_i + V_i > U(\boldsymbol{s})$. But since V is generally unknown, all techniques for focusing computations on a neighborhood of this optimal path must instead rely on some "heuristic underestimate"

$$\varphi_i \;=\; \varphi(\boldsymbol{x}_i) \;\leq\; V_i. \tag{2.3}$$

A stronger *"consistency"* requirement is often imposed instead:

$$\varphi_j \;\leq\; C_{ij} + \varphi_i; \qquad \forall\, i, j. \tag{2.4}$$

(Note that $\varphi \equiv V$ is the maximum among all consistent heuristics that also satisfy $\varphi(\boldsymbol{s}) = 0$.)

Such consistent underestimates are readily available for geometrically embedded graphs. Suppose $X \subset \mathbb{R}^n$ and $d_{ij} = \left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|_2$. If the "maximum speed" $F_2 > 0$ is such that $C_{ij} \geq d_{ij}/F_2$ for all i and j, then $\varphi_i = \|\boldsymbol{x}_i - \boldsymbol{s}\|_2 / F_2 \leq V_i$. On a Cartesian grid-type graph, the Manhattan distance provides a better (tighter) underestimate $\varphi_i = \|\boldsymbol{x}_i - \boldsymbol{s}\|_1 / F_2$. For more general embedded graphs, a much better underestimate φ can be produced by *"Landmark sampling"* [28], but this requires additional precomputation and increases the memory footprint of the algorithm.

Some algorithms for this problem also rely on "heuristic overestimates"

$$\psi_i \;=\; \psi(\boldsymbol{x}_i) \;\geq\; V_i.$$

An overestimate can be obtained as a total cost of *any* path from $\boldsymbol{x}_i$ to $\boldsymbol{s}$ or can also be found using landmark precomputations [28]. For structured geometrically embedded graphs, an analytic expression might also be available. E.g., on a Cartesian grid, if the "minimum speed" $F_1 > 0$ is such that $C_{ij} \leq d_{ij}/F_1$, we can use $\psi_i = \|\boldsymbol{x}_i - \boldsymbol{s}\|_1 / F_1$.

2.2.2 A* Restriction Techniques

A* techniques restrict computations to potentially relevant nodes by limiting the number of nodes that become CONSIDERED. A more accurate φ restricts a larger number of nodes from becoming CONSIDERED, and if $\varphi = V$ then only those nodes actually on the $\boldsymbol{s} \to \boldsymbol{t}$ optimal path are ever ACCEPTED.

Standard A* (SA*). This version of A* is the one most often described in the literature [32]. Unlike in Dijkstra's algorithm, the CONSIDERED nodes are sorted and ACCEPTED based on $(U_i + \varphi_i)$ values. This change affects line 6 in our pseudocode. The resulting algorithm typically ACCEPTS far fewer nodes before terminating: irrelevant nodes with large φ values might still become CONSIDERED (if their neighbors are accepted) but will have lower priority and most of them will never become ACCEPTED themselves. Moreover, the consistency of φ ensures that ACCEPTED nodes receive exactly the same values as would have been produced by the original Dijkstra's method. If $\boldsymbol{x}_i$ actually depends on $\boldsymbol{x}_j \in N_i^+$, then

$$U_i = C_{ij} + U_j \quad \Longrightarrow \quad U_i \geq \left(\varphi_j - \varphi_i\right) + U_j \quad \Longleftrightarrow \quad U_i + \varphi_i \geq U_j + \varphi_j,$$

guaranteeing that under SA* $\boldsymbol{x}_i$ will not be ACCEPTED before $\boldsymbol{x}_j$.

Alternative A* (AA*). A less common variant of A* is described in [8]. Instead of favoring nodes with small φ, AA* simply ignores nodes that are clearly irrelevant. AA* relies on an underestimate φ (no longer required to satisfy (2.4)) and an additional upper bound $\Psi \geq U(\boldsymbol{s})$. (If an analytic or precomputed ψ is available, we can take $\Psi = \psi(\boldsymbol{t})$. But it is also possible to use the total cost of any feasible path from $\boldsymbol{s}$ to $\boldsymbol{t}$.)

During Dijkstra's algorithm, a node $\boldsymbol{x}_i$ with $U_i + \varphi_i > \Psi$ (hence $U_i + V_i > \Psi$) is surely not a part of the optimal path. Thus, to speed up Dijkstra's algorithm, in AA* we still sort CONSIDERED nodes based on U values, but on line 11 we only mark $\boldsymbol{x}_i$ CONSIDERED if

$U_i + \varphi_i \leq \Psi$. Since the order of acceptance is the same, it is clear that AA* produces the same values as Dijkstra's, but the efficiency of this technique is clearly influenced by the quality of Ψ (the smaller it is, the smaller is the number of CONSIDERED nodes). This reliance on Ψ is a downside (since SA* only needs φ), but has the advantage of making AA* also applicable to the label-correcting methods [8]. In section §2.3 we argue that AA* is also more suitable for continuous optimal control problems, in which an Ψ is often readily available.

AA* with Branch & Bound (B&B). In AA* Ψ remains static throughout the algorithm. The idea of *Branch & Bound* (B&B) is to dynamically decrease Ψ as we gain more information about the graph, making use of an overestimate function ψ. When ACCEPTING a node $\boldsymbol{x}_i$, we can also set

$$\Psi \leftarrow \min\{\Psi,\ U_i + \psi_i\}.$$

Exact estimates. Using "exact estimates" with A* would result in the maximal domain restriction. For both A* techniques, if $\varphi \equiv V$ and $\Psi \equiv U(\boldsymbol{s})$, the algorithm would only ACCEPT the nodes lying on an optimal path.

2.3 Domain Restriction in a Continuous Setting

The continuous time-optimal isotropic control problem deals with minimizing the time-to-exit for a vehicle, whose dynamics is governed by

$$\begin{cases} \dot{\boldsymbol{y}}(t) &= f(\boldsymbol{y}(t))\,\boldsymbol{a}(t), \\ \boldsymbol{y}(0) &= \boldsymbol{x} \in \Omega \subset \mathbb{R}^n, \end{cases} \tag{2.5}$$

Here $\boldsymbol{x}$ is the starting position, $\boldsymbol{a}(t) \in S^{n-1}$ is the control (i.e., the direction of motion) chosen at the time t, $\boldsymbol{y}(t)$ is the vehicle's time-dependent position, and f is the spatially-dependent speed of motion. We will further assume the existence of two constants F_1

and F_2 such that $0 < F_1 \leq f(\boldsymbol{x}) \leq F_2$ holds $\forall\, \boldsymbol{x} \in \bar{\Omega}$. For every time-dependent control $\boldsymbol{a}(\cdot)$ we define the total time to the exit set $Q \subseteq \partial\Omega$ as $T_{\mathbf{x},\mathbf{a}} = \min\{t \geq 0 \mid \boldsymbol{y}(t) \in Q\}$. The *value function* $u : \Omega \to [0, +\infty)$ is then naturally defined as

$$u(\boldsymbol{x}) \;=\; \inf_{\boldsymbol{a}(\cdot)} \{T_{\mathbf{x},\mathbf{a}} \;+\; q\left[\boldsymbol{y}\left(T_{\mathbf{x},\mathbf{a}}\right)\right]\},$$

where $q : Q \to [0, +\infty)$ is the exit-time penalty. Bellman's optimality principle can be used to show that, if u is a smooth function, it must satisfy a static Hamilton-Jacobi-Bellman PDE

$$\min_{\boldsymbol{a} \in A} \{(\nabla u(\boldsymbol{x}) \cdot \boldsymbol{a})\, f(\boldsymbol{x}) \;+\; 1\} \;=\; 0,$$

with the natural boundary condition $u = q$ on Q. Using the isotropic nature of the dynamics, it is clear that the minimizer (i.e., the optimal initial direction of motion starting from $\boldsymbol{x}$) is $\boldsymbol{a}_* = -\nabla u(\boldsymbol{x})/\|\nabla u(\boldsymbol{x})\|$ and the equation is equivalent to the Eikonal PDE (3.1). If the value function u is not smooth, it can still be interpreted as a unique *viscosity solution* of this PDE [19].

Solving this PDE to recover the value function is the key idea of the *dynamic programming*. An analytic solution is usually unavailable, so numerical methods are needed to approximate u. We use a first-order upwind discretization, whose monotonicity and consistency yield convergence to the viscosity solution [5]. To simplify the notation, we describe everything on a cartesian grid in $\mathbb{R}^2$, though higher dimensional generalizations are straightforward and similar discretizations are also available on simplicial meshes (e.g., [41, 69]; see also Figure 2.4). We will assume

- $\bar{\Omega} = [0, 1] \times [0, 1]$ is discretized on a $m \times m$ uniform grid X with spacing $h = 1/(m - 1)$.
- A *gridpiont* or *node* is denoted by $\boldsymbol{x}_{ij}$ with corresponding *value* $U_{ij} = U(\boldsymbol{x}_{ij}) \approx$

$u(\boldsymbol{x}_{ij})$, *speed* $f_{ij} = f(\boldsymbol{x}_{ij})$, and *neighbors*

$$N_{ij} \;=\; N(\boldsymbol{x}_{ij}) \;\triangleq\; \left\{\boldsymbol{x}_{i-1,j},\, \boldsymbol{x}_{i+1,j},\, \boldsymbol{x}_{i,j-1},\, \boldsymbol{x}_{i,j+1}\right\}.$$

This notation will be slightly abused (e.g., a gridpoint $\boldsymbol{x}_i$ with a corresponding value U_i, speed f_i, etc...) whenever we emphasize the ordering of gridpoints rather than their geometric position.

- We will assume that $\mathcal{Q} \subseteq \partial\Omega$ is *well-discretized* on the grid, and we define the *discretized exit-set* $Q = \mathcal{Q} \bigcap X$.

  In particular, the focus of our computational experiments will be on the case $\mathcal{Q} = Q = \{\boldsymbol{t}\}$ with the exit time-penalty $q(\boldsymbol{t}) = 0$. We note that $\boldsymbol{t}$ does not have to be on the boundary of the square $\bar{\Omega}$: if $\boldsymbol{t} \in (0,1)^2$, then $\Omega = (0,1)^2 \backslash \{\boldsymbol{t}\}$, and the border of the square is treated as an essentially outflow boundary. This corresponds to solving a $\bar{\Omega}-$*constrained* optimal control problem, with u interpreted as a constrained viscosity solution [4].

We use the upwind finite differences [66] to approximate the derivatives of (3.1), resulting in a system of discretized equations. Using the standard four-point nearest-neighbors stencil at each $\boldsymbol{x}_{ij} \in X$, this results in:

$$\left(\max\left\{D^{-x}U_{ij}, -D^{+x}U_{ij}, 0\right\}\right)^2 \;+\; \left(\max\left\{D^{-y}U_{ij}, -D^{+y}U_{ij}, 0\right\}\right)^2 \;=\; \frac{1}{f_{ij}^2}, \tag{2.6}$$

$$\text{where} \quad u_x(x_i, y_j) \approx D^{\pm x}U_{ij} = \frac{U_{i\pm1,j} - U_{ij}}{\pm h}, \quad \text{and} \quad u_y(x_i, y_j) \approx D^{\pm y}U_{ij} = \frac{U_{i,j\pm1} - U_{ij}}{\pm h}.$$

If all the neighboring values are known, this is really a “quadratic equation in disguise" for U_{ij}. Letting $U_H = \min\left\{U_{i-1,j}, U_{i+1,j}\right\}$ and $U_V = \min\left\{U_{i,j-1}, U_{i,j+1}\right\}$ reduces (2.6) to

$$\left(U_{ij} - U_H\right)^2 \;+\; \left(U_{ij} - U_V\right)^2 \;=\; \frac{h^2}{f_{ij}^2}, \tag{2.7}$$

provided the solution satisfies $U_{ij} \geq \max\left\{U_H, U_V\right\}$; otherwise we perform a *one-sided update*:

$$U_{ij} \;=\; \min\{U_H, U_V\} \;+\; \frac{h}{f_{ij}}. \tag{2.8}$$

The system of discretized equations ((2.7) and (2.8) for all (i, j)) are *monotone causal* since U_{ij} needs only its *smaller* neighboring values to produce an update.

Sethian's **Fast Marching Method (FMM)** [67] and another Dijkstra-like algorithm [81] due to Tsitsiklis take advantage of this monotone causality. FMM can be obtained from Dijkstra's Method by changing the lines 3 and 7 to instead use the continuous update procedure (equations (2.7) and (2.8)). Similarly to Dijkstra's method, FMM computes the value function on the entire grid in $O(M \log M)$ operations, where $M = m^2$ is the number of gridpoints. The key question is whether a significant reduction of computational cost is possible if we are only interested in an optimal trajectory starting from a single (pre-specified) source gridpoint $\boldsymbol{s}$.

Remark 3. *Restricting FMM to a smaller (relevant) subset of* Ω *via A*-techniques is precisely the focus of this paper. But a legitimate related question is whether the dynamic programming approach is at all necessary when a single trajectory is all that we desire?*

In contrast to path planning on graphs, in the continuous control community, optimal trajectories for single source problems are typically recovered via ***Pontryagin Maximum Principle (PMP)*** *[63]. This involves solving a two point boundary value problem for a state-costate system of ODEs, which in our context could be also derived as characteristic ODEs of the Eikonal PDE* (3.1)*. One advantage of using PMP is that, unlike the dynamic programming, it does not suffer from the* curse of dimensionality*. In higher dimensions, solving a two-point boundary value problem is much more efficient than solving a PDE on the whole domain. Unfortunately, PMP is harder to apply if the speed function* f *is not smooth. Even more unpleasantly, depending on the initial*

guess used to solve the two-point boundary value problem, that method often converges to locally *optimal trajectories. In contrast, the dynamic programming always yields a globally optimal trajectory, and our approach can be used to lower its computational cost in higher dimensions. In fact, we show that both techniques can be used together, with a prior use of PMP improving the efficiency of A*, and A* verifying the global optimality of a PMP-produced trajectory.*

2.3.1 Domain restriction without heuristic underestimates.

Our algorithmic goal is to restrict FMM to a dynamically defined subset of the grid using underestimates of the cost-to-go and the previously computed values. This is the essence of several A*-type techniques compared in sections 2.3.2-2.3.5. But to motivate the discussion, we start by considering several simpler domain restriction techniques that do not involve the run-time use of underestimates.

First, we note that FMM can be terminated immediately after the gridpoint $\boldsymbol{s}$ is ACCEPTED. In practice, this is unlikely to yield significant computational savings unless the set

$$L \;=\; \{\boldsymbol{x} \in \bar{\Omega} \;|\; u(\boldsymbol{x}) \leq u(\boldsymbol{s})\}$$

is much smaller than the entire $\bar{\Omega}$. (E.g., see the bolded level set ∂L in Figure 2.3A.)

Second, it is possible to use a "bi-directional FMM" (similar to the bi-directional Dijkstra's [61]) by expanding two ACCEPTED clouds from the source and the target and stopping the process when they meet. The first gridpoint accepted in both clouds is guaranteed to lie on an $O(h)$-suboptimal trajectory from $\boldsymbol{s}$ to $\boldsymbol{t}$. This approach is potentially much more efficient than the above. E.g., for a constant speed function $f = 1$, it cuts the n-dimensional volume of the ACCEPTED set by the factor of 2^{n-1}; see Figure 2.2.

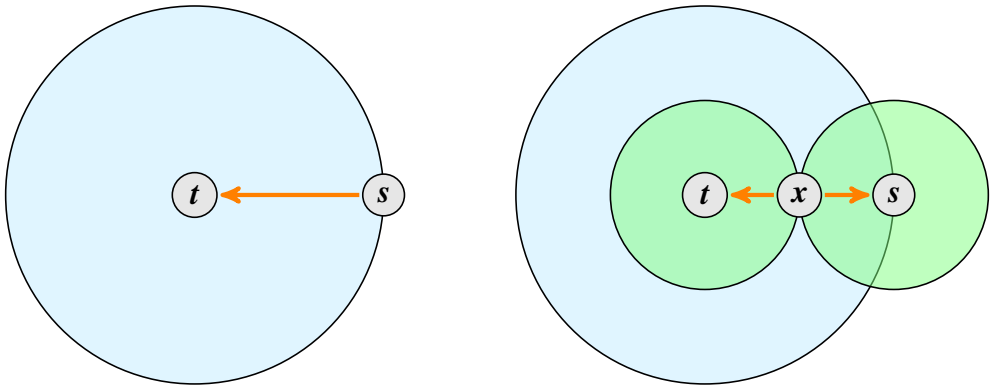


Figure 2.2: FMM expands computations outwards from $\boldsymbol{t}$, shown by the large circle. The two smaller circles each expand from $\boldsymbol{t}$ and $\boldsymbol{s}$ and represent the computations performed during BiFMM. In this simple situation BiFMM considers 50% of the domain that FMM considers. To recover the global optimal trajectory from $\boldsymbol{s}$ to $\boldsymbol{t}$ using BiFMM one must recover the optimal trajectories from $\boldsymbol{x}$ to $\boldsymbol{t}$ and $\boldsymbol{x}$ to $\boldsymbol{s}$ and join them together.

Third, a different "elliptical restriction" approach is also applicable (and can be combined with the above bidirectional technique) whenever an overestimate for the minimal time from $\boldsymbol{s}$ to $\boldsymbol{t}$ is available.

Lemma 1. *Suppose the exit-set is given by a single target point* $\boldsymbol{t}$, $d = |\boldsymbol{s} - \boldsymbol{t}|$, *and* Ψ *is a known constant such that* $\Psi \geq u(\boldsymbol{s})$. *Then the optimal trajectory* $\boldsymbol{y}(\cdot)$ *satisfying* (2.5) *from* $\boldsymbol{y}(0) = \boldsymbol{s}$ *is contained within the prolate spheroid* $E(\boldsymbol{s}, \boldsymbol{t})$ *(an ellipse in 2D) satisfying*

$$\textit{Foci} = \{\boldsymbol{s}, \boldsymbol{t}\} \qquad \textit{and} \qquad \begin{cases} \textit{Major semi-axis} \;=\; a \;=\; \dfrac{F_2\Psi}{2}, \\[2ex] \textit{Minor semi-axis} \;=\; b \;=\; \dfrac{1}{2}\sqrt{F_2^2\Psi^2 - d^2}. \end{cases} \tag{2.9}$$

Proof. Let d^* and T^* be the distance and time along the optimal trajectory from $\boldsymbol{s}$ to $\boldsymbol{t}$. Then

$$\frac{d^*}{F_2} \;\leq\; T^* \;\leq\; \Psi, \tag{2.10}$$

For any $\boldsymbol{x}$ along the optimal trajectory we have

$$|\boldsymbol{x} - \boldsymbol{s}| + |\boldsymbol{t} - \boldsymbol{x}| \;\leq\; d^* \;\leq\; F_2\Psi.$$

This inequality defines a prolate spheroid in $\mathbb{R}^n$ and (2.9) immediately follows. □

Even if we are interested in an unconstrained problem (find the quickest $(\boldsymbol{s}, \boldsymbol{t})$ trajectory in $\mathbb{R}^2$), finite computer memory forces us to solve a *state-constrained* problem instead (find the quickest $(\boldsymbol{s}, \boldsymbol{t})$ trajectory contained in $\bar{\Omega}$). The above Lemma is thus also useful to answer a related question: for which starting points $\boldsymbol{s}$ does the $\bar{\Omega}$-constrained problem have the same value function as the unconstrained? Clearly, for any point $\boldsymbol{s}$ such that $E(\boldsymbol{s}, \boldsymbol{t}) \subset \bar{\Omega}$, enlarging the domain would not decrease $u(\boldsymbol{s})$.

Higher dimensional savings. Restricting computations to $E(\boldsymbol{s}, \boldsymbol{t})$ has an increasing effect in higher dimensions. The fraction $\mathcal{P}$ of the volume of $E(\boldsymbol{s}, \boldsymbol{t})$ to the volume of the smallest bounding rectangular box B is given by $\mathcal{P} = \frac{\pi^{n/2}}{2^n \Gamma(n/2+1)}$, which quickly approaches zero as n grows. (E.g., in $\mathbb{R}^2$ this fraction is $(\pi/4) \approx 78.5\%$, while in $\mathbb{R}^6$ it is already $\approx 8\%$.) If $\bar{\Omega} = B$, the restriction to $E(\boldsymbol{s}, \boldsymbol{t})$ yields the computational savings of $(1 - \mathcal{P})$; the savings are even higher if $\bar{\Omega}$ is any other box-rectangular domain fully containing $E(\boldsymbol{s}, \boldsymbol{t})$.

Formulas for Ψ can be naturally obtained by computing (or bounding from above) the time along any feasible path from $\boldsymbol{s}$ to $\boldsymbol{t}$. On a convex domain Ω, the most obvious choice is $\Psi_1 = d/F_1$ (i.e., follow the straight line from $\boldsymbol{s}$ to $\boldsymbol{t}$ at the minimum speed F_1). For problems with the unit speed of motion, $f(\boldsymbol{x}) = 1 = F_1 = F_2$, $\Psi_1 = d$, and the ellipse collapses to a straight line segment.

A more accurate overestimate can be obtained by computing the exact time needed

to traverse that straight line trajectory:

$$\Psi_2 \;=\; \int_0^{|t-s|} \frac{dr}{f\left(s + \frac{t-s}{|t-s|}r\right)} \;=\; \int_0^1 \frac{|t-s|}{f\left(s + (t-s)r\right)} dr \;\leq\; \Psi_1.$$

For non-convex domains, a similar upper bound can be obtained by integrating the slowness $1/f$ along any feasible trajectory (e.g., the shortest $\bar{\Omega}$-constrained path from s to t).

Finally, we will also consider the third ("ideal") option, with $\Psi_3 = u(s) \leq \Psi_2$. While practically unattainable, Ψ_3 is useful to illustrate the upper bound on efficiency of various domain restriction techniques. In practice, it can be approximated by using $U(s)$ precomputed on a coarser grid or using the output of Pontryagin-Maximum-Principle-based computations (see the example in section 2.4.4). In the latter case, the techniques discussed in this paper can be viewed as a method for verifying the global optimality of a known locally-optimal trajectory. Figure 2.3A shows the (s, t)-focused ellipses for a specific example with a highly oscillatory speed function.

2.3.2 Dynamic domain restriction: underestimates and A*-techniques.

The previous subsection described a priori domain restriction techniques. Here, our goal is to further restrict the computations *dynamically* by using the solution already computed on parts of $\bar{\Omega}$. The actual viscosity solution $u(s)$ depends only on values along a characteristic (i.e., an optimal (s, t) trajectory). Ideally, we would like to compute the numerical solution U only for the gridpoints within an immediate neighborhood of that trajectory, potentially yielding a much greater speedup than the techniques described above.

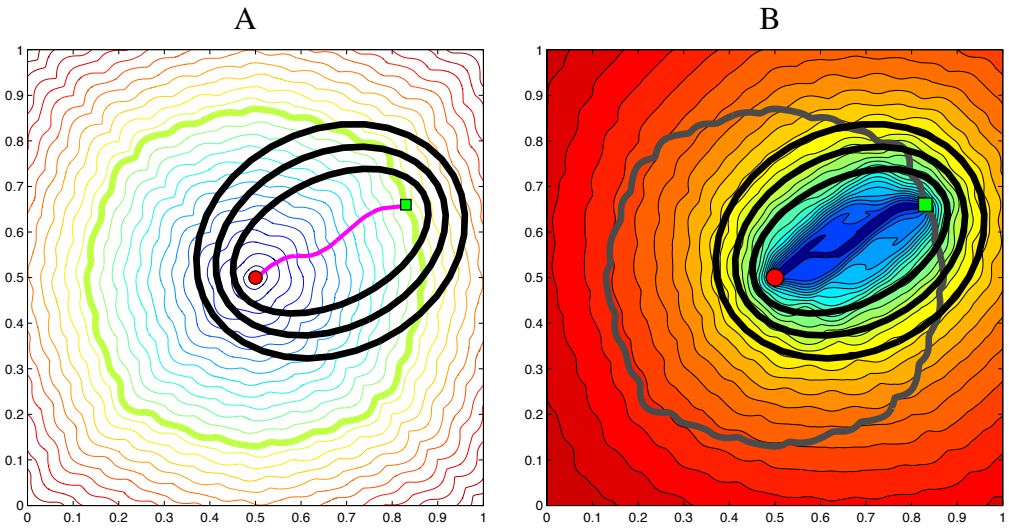


Figure 2.3: **A.** Level sets of U computed with a highly oscillatory speed $f(x, y) = 2 + 0.5\sin(20\pi x)(20\pi y)$. The curve ∂L is indicated by a thicker contour line. Three ellipses corresponding to Ψ_1, Ψ_2, and Ψ_3 are shown in black. **B.** the level sets of $\log_{10}[U(\boldsymbol{x}) + V(\boldsymbol{x}) - U(\boldsymbol{s}) + 0.01]$ for the same problem.

Consider a function $v(\boldsymbol{x})$ specifying the min-time from $\boldsymbol{s}$ to $\boldsymbol{x}$. (It is easy to see that v is also a viscosity solution of the Eikonal PDE (3.1), but with the different boundary condition $v(\boldsymbol{s}) = 0$.) We note that $u(\boldsymbol{x}) + v(\boldsymbol{x}) \geq u(\boldsymbol{s}) = v(\boldsymbol{t})$ for all $\boldsymbol{x} \in \Omega$, and this becomes an equality if and only if $\boldsymbol{x}$ lies on an optimal $(\boldsymbol{s}, \boldsymbol{t})$ trajectory. (See the level sets of $u + v$ in Figure 2.3B.) Since v is generally unknown, any practical restriction of computational domain will have to rely on an "admissible underestimate heuristic" φ, satisfying $\varphi(\boldsymbol{x}) \leq v(\boldsymbol{x})$, $\forall\, \boldsymbol{x} \in \bar{\Omega}$. As we will see, tighter underestimates result in more efficient domain restrictions. Here we enumerate several natural heuristic underestimates:

1. *Naïve heuristic* is obtained by assuming the maximum speed of travel along the straight line:

$$\varphi^0(\boldsymbol{x}) \;=\; |\boldsymbol{s} - \boldsymbol{x}| \;/\; F_2. \tag{2.11}$$

Several papers on SA* versions of FMM [26, 55, 84, 85] have relied on its scaled-down version $\varphi^0_\lambda = \lambda\varphi^0(\boldsymbol{x})$ with $\lambda \in [0, 1]$.

2. *Coarse grid heuristic* [57, 59] is based on precomputing V on a coarser $(Rm) \times (Rm)$ grid with $R \in (0, 1)$. If we use V^R to denote the interpolation of that solution on X, the heuristic is then defined as $\varphi^C_{\lambda,R} = \lambda V^R$, where $\lambda \in [0, 1]$ is chosen to ensure that the result is a true underestimate.

3. *Landmarking-based heuristic* [58, 59] is a continuous version of the landmarking technique on graphs [28]. This relies on pre-computing/storing the minimum time from every node to a number of "landmarks"; the triangle inequality is then used to obtain the lower bound $\varphi^L(\boldsymbol{x}) \leq v(\boldsymbol{x})$. The high computational cost and memory footprint make this approach useful for repeated queries only. (I.e., only if the optimal trajectory problem has to be solved for many different $(\boldsymbol{s}, \boldsymbol{t})$ pairs.)

4. *Higher-speed heuristic* can be obtained by starting with a special speed-overestimate $f_0(\boldsymbol{x}) \geq f(\boldsymbol{x})$, such that the corresponding value function $v_0(\boldsymbol{x}) \leq v(\boldsymbol{x})$ is known analytically, and then setting $\varphi(\boldsymbol{x}) = v_0(\boldsymbol{x})$. (Note that (2.11) can be also derived this way by taking $f_0(\boldsymbol{x}) \equiv F_2$.) If the $(\boldsymbol{s}, \boldsymbol{t})$ path-planning has to be performed for *many different* speed functions f, the above approach can be useful even if v_0 has to be approximated numerically. One such example is included in section 2.4.4.

5. *Scaled "Oracle" heuristic* [59] is defined as $\bar{\varphi}_\lambda(\boldsymbol{x}) = \lambda v(\boldsymbol{x})$ with $\lambda \in [0, 1]$. This is clearly not a practical underestimate, but a theoretical device useful in studying the accuracy/efficiency tradeoffs of various domain restriction techniques. Since v is generally unavailable, our benchmarking relies on a numerical approximation; i.e., $\bar{\varphi}_\lambda(\boldsymbol{x}) = \lambda V(\boldsymbol{x})$, where V is (pre-)computed on the same grid X.

The first of these (the Naïve heuristic) is a conservative underestimate that is cheaply available for all problems – including the situations with discontinuous speed functions and/or non-convex domains. The other underestimates are more expensive to produce,

but usually result in a more significant domain restriction. Thus, their use is particularly justified when the same speed function is used repeatedly to solve numerous (single source / single target) problems.

We emphasize that this paper is in a sense "underestimate-neutral." A good underestimate is obviously important, but our focus is on how it should be used rather than on how to build it.

Continuous A* Techniques. Both SA* and AA* algorithms on graphs may be easily adapted to the continuous setting using any of the above heuristics. Just like on graphs, SA*-FMM increases the "processing-priority" of nodes with low φ values, whereas our AA*-FMM avoids considering nodes guaranteed not to be a part of any $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory. Each of these methods successfully restricts the computations, but with different trade-offs between the execution time, memory footprint, amount of restriction, and computational error.

2.3.3 Prior work on SA*-FMM.

From the implementation standpoint, SA*-FMM is fairly straightforward. It requires modifying a single line 6 of FMM (see Dijkstra's algorithm): ACCEPT the node with minimal $U + \varphi$. (Since $u + v$ is minimal along the $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory, $U(\boldsymbol{x}) + \varphi(\boldsymbol{x})$ is used to indicate how close $\boldsymbol{x}$ is to that trajectory.) However, the analysis of this method's output is more subtle.

On graphs, the consistency of the heuristic underestimate (i.e., the condition (2.4)) guarantees that all SA*-accepted nodes receive the same values as would have been produced by Dijkstra's. In contrast, SA*-FMM exhibits a performance tradeoff based on

whether φ satisfies a more restrictive and stencil-dependent consistency condition (defined below). If φ is inconsistent, some of the gridpoints may be ACCEPTED prematurely, resulting in additional numerical errors. On the other hand, if φ is consistent, the efficiency of the domain restriction is significantly decreased, and this restricted domain does not shrink to zero volume as $h \to 0$.

The presence of additional errors might seem counterintuitive. After all, if an $(\boldsymbol{x}_i, \boldsymbol{t})$-optimal trajectory passes through some $\boldsymbol{x}_j$, then, for φ defined by formula (2.11),

$$u(\boldsymbol{x}_i) \;=\; (\text{Time from } \boldsymbol{x}_i \text{ to } \boldsymbol{x}_j) \;+\; u(\boldsymbol{x}_j) \;\geq\; \frac{\left|\boldsymbol{x}_j - \boldsymbol{x}_i\right|}{F_2} \;+\; u(\boldsymbol{x}_j) \;\geq\; \varphi(\boldsymbol{x}_j) - \varphi(\boldsymbol{x}_i) \;+\; u(\boldsymbol{x}_j),$$

guaranteeing that $u(\boldsymbol{x}_i)+\varphi(\boldsymbol{x}_i) \geq u(\boldsymbol{x}_j)+\varphi(\boldsymbol{x}_j)$. Turning to numerical solutions, we would hope for the same argument to work for U_i and U_j, and indeed it does if U_i is computed by a one-sided update formula (2.8). But for a first-order upwind discretization in $\mathbb{R}^2$, a generic gridpoint $\boldsymbol{x}_i$ depends on 2 other gridpoints that straddle $\boldsymbol{x}_i$'s characteristic. To produce the same numerical values under SA*-FMM and FMM, we would need to know that $U_i + \varphi(\boldsymbol{x}_i) \geq U_j + \varphi(\boldsymbol{x}_j)$ whenever $\boldsymbol{x}_i$ directly depends on $\boldsymbol{x}_j$.

Suppose there exists a constant $\lambda > 0$ such that

$$U_i \text{ directly depends on } U_j \qquad \Longrightarrow \qquad U_i \;>\; U_j + \lambda|\boldsymbol{x}_i - \boldsymbol{x}_j|, \qquad\qquad \forall\, i, j. \quad (2.12)$$

The proper ordering is then guaranteed provided the underestimate φ satisfies the consistency condition

$$|\varphi(\boldsymbol{x}_i) - \varphi(\boldsymbol{x}_j)| \;\leq\; \lambda|\boldsymbol{x}_i - \boldsymbol{x}_j|, \qquad\qquad \forall\, i, j, \qquad\qquad (2.13)$$

which is easy to ensure by using the underestimate

$$\varphi^0_\lambda(\boldsymbol{x}) \;=\; \lambda\varphi^0(\boldsymbol{x}).$$

Unfortunately, the condition (2.12) is stencil-dependent and in this section we explore its implications both on grids and triangular meshes; see Figure 2.4.

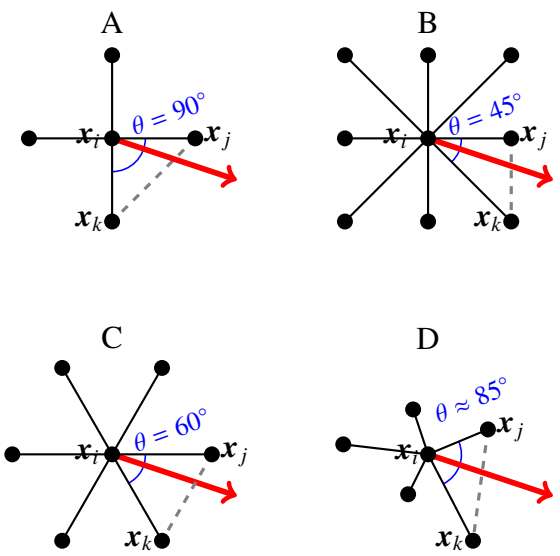


Figure 2.4: Four computational stencils for a node $\boldsymbol{x}_i$: four-point and eight-point stencils on a Cartesian grid (A and B), a six-point stencil on a regular triangular mesh (C), and a five-point stencil on a unstructured triangular mesh (D).

Suppose that $\boldsymbol{x}_i$'s characteristic is straddled by $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$, where θ is the angle $\angle \boldsymbol{x}_j \boldsymbol{x}_i \boldsymbol{x}_k$ and γ is the angle between the characteristic and $\boldsymbol{x}_i \boldsymbol{x}_j$. Since for the Eikonal equation the characteristics coincide with gradient lines and our numerical approximation is piecewise linear, it is easy to show that[1]

$$(U_i - U_j) \;=\; \cos\gamma \, |\boldsymbol{x}_i - \boldsymbol{x}_j| \,/\, f(\boldsymbol{x}_i) \;\geq\; \cos(\theta) h \,/\, f(\boldsymbol{x}_i).$$

The latter lower bound is actually sharp when the characteristic is parallel to $\boldsymbol{x}_i \boldsymbol{x}_k$ and $h = |\boldsymbol{x}_i - \boldsymbol{x}_j|$. This means that

- $\varphi_0^0 \equiv 0$ is the only consistent underestimate for stencil 2.4A (i.e., $\lambda = \frac{1}{F_2} \cos\frac{\pi}{2} = 0$). Thus, SA*-FMM will usually result in additional errors.
- for stencil 2.4B, φ_λ^0 becomes consistent for $\lambda \leq \frac{1}{F_2 \sqrt{2}} = \frac{\cos\frac{\pi}{4}}{F_2}$.
- for a local stencil used on a general triangular mesh (e.g., Figure 2.4D), if $\bar{\theta} < \frac{\pi}{2}$

[1] We note that this observation was previously used in [83] to find the conditions for applicability of Dial-like algorithms.

is an upper bound on angles θ present in the mesh, then φ^0_λ becomes consistent for $\lambda \leq \cos(\bar{\theta})/F_2$.

Interestingly, the importance of consistency conditions for SA*-FMM was only recently recognized in [84, 85], while all the prior versions treated this in an ad-hoc fashion. To summarize:

- [2005] Ferguson & Stentz [26] adapt D* algorithms to continuous optimal trajectory problems discretized on stencil 2.4B. They also introduce an SA*-type technique within D* to further improve the performance. The method relies on φ^0_λ to ensure the right order of gridpoint processing, but the choice of λ is never explained explicitly.
- [2005, 2006, 2008] Peyré & Cohen [57, 58, 59] adapt SA* for FMM on stencil 2.4A using underestimates $\varphi^C_{\lambda,R}$ and φ^L. The authors acknowledge that their version of SA*-FMM produces additional errors and experimentally study the dependence of these errors on the tightness of underestimates. However, they do not analyze the behavior of errors under grid refinement.
- [2007] Pêtrès [55] defines an SA*-FMM variant on a stencil 2.4A with φ^0_λ. A brief description of a bi-directional version of SA*-FMM is also included. Pêtrès acknowledges that, for large λ, the additional (SA*-induced) errors can be larger than discretization errors, but does not analyze how that ratio changes under grid refinement.
- [2011, 2012] Yershov & LaValle [84, 85] use FMM with acute triangular meshes as in [69] in $\mathbb{R}^2$, $\mathbb{R}^3$, and on two dimensional manifolds. Their problems of interest use $f \equiv 1$ on a domain with obstacles. The authors use SA*-FMM with φ^0_λ and prove that $\lambda = \cos(\bar{\theta})$ guarantees absence of additional errors. The authors

state that in their experiments SA*-FMM processed only 50% of the gridpoints processed by FMM.

Remark 4. *Every implementation of SA*-FMM also involves an "efficiency versus memory footprint" tradeoff. Since the binary heap of* CONSIDERED *nodes is sorted based on $U + \varphi$, every heap-maintenance operation relies on availability of $\varphi(\boldsymbol{x})$ for many nodes on the heap. This happens whenever a* FAR *node becomes* CONSIDERED*, or a* CONSIDERED *node receives a smaller value or becomes* ACCEPTED*. If φ is re-computed each time it is needed (e.g., by* (2.11)*), this introduces a noticeable overhead to each heap operation. An alternative (to cache φ the first time it is computed for each* CONSIDERED *node) is certainly more efficient, but significantly increases the memory footprint, particularly on larger grids and in higher-dimensional problems. In Section 2.4 we include the performance data for both of these approaches.*

2.3.4 Accuracy or efficiency?

The errors introduced by any A*-type restriction techniques are not very surprising once we recall that the numerical viscosity of the discretization results in a large domain of computational dependency for $U(\boldsymbol{s})$. To formalize this argument, we will consider a *dependency digraph* G built on the nodes of X. For $\boldsymbol{x}_i$ and $\boldsymbol{x}_j \in N_i$, G includes an arc $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ if $\boldsymbol{x}_i$ *directly depends* on $\boldsymbol{x}_j$; i.e., if U_j is needed to compute U_i. We will say that $\boldsymbol{x}_i$ *depends* on $\boldsymbol{x}_j$ if there exists a path in G from $\boldsymbol{x}_i$ to $\boldsymbol{x}_j$. Due to the monotone causality of the upwinding discretization, this dependence implies $U_i > U_j$; thus, G is acyclic and every path on it leads to $\boldsymbol{t}$. We will also use $G(\boldsymbol{s})$ to denote the subset of G reachable from $\boldsymbol{s}$.

Consider *any* domain restriction technique that results in accepting only nodes from

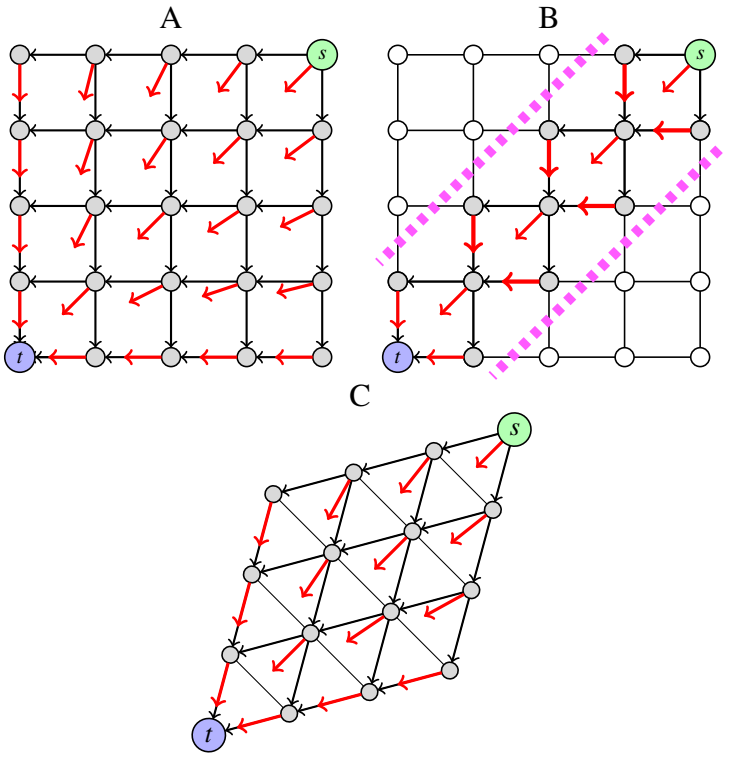


Figure 2.5: Domain restriction for the constant speed example. The full dependency graph is shown for a 4-point stencil on a Cartesian grid (A) and for a 6-point stencil on a triangular mesh (C). Thin black arrows show the arcs of $G(\boldsymbol{s})$. The shorter arrows show the characteristic direction for each node. Subfigure (B) shows a domain-restricted computation. The nodes inside of the dashed lines represent the nodes that pass the A* condition (2.14). The thicker characteristic arrows highlight the "optimal" directions that have changed due to this domain restriction.

some $\hat{X} \subset X$ and produces some numerical approximation of the value function $U^*(\boldsymbol{x})$ for each $\boldsymbol{x} \in \hat{X}$. If $G(\boldsymbol{s}) \not\subset \hat{X}$, we cannot expect $U^*(\boldsymbol{s})$ to be the same as $U(\boldsymbol{s})$ produced by FMM on the full X. In other words, if we insist on avoiding *any* additional (restriction-induced) errors, this typically results in severe constraints on the efficiency of the domain restriction.

To illustrate this point, we consider a very simple problem with $\boldsymbol{t}$ and $\boldsymbol{s}$ in opposite corners of $\bar{\Omega}$; see Figure 2.5A. With $f \equiv 1$ the optimal trajectory from every starting

position $\boldsymbol{x}$ is just a straight line to $\boldsymbol{t}$. But it is easy to see that $G(\boldsymbol{s})$ includes all nodes in X; thus, **any** restriction will result in $U^*(\boldsymbol{s}) > U(\boldsymbol{s})$. We emphasize that this phenomenon has nothing to do with the non-existence of consistent φ for the 4-point stencil discretization on a cartesian grid. Figure 2.5C shows an equivalent example on a regular triangular mesh. As explained in [84, 85], taking $\lambda = 1/2$ will ensure that φ^0_λ is consistent for this problem and stencil. As a result, SA*-FMM will produce $U^*(\boldsymbol{s}) = U(\boldsymbol{s})$, but at the cost of accepting exactly the same set of nodes[2] as FMM (i.e., $\hat{X} = X$).

For these reasons, we believe that asking for $U^*(\boldsymbol{s}) = U(\boldsymbol{s})$ on every fixed grid is unrealistic and makes the domain restriction much less efficient. A more attractive strategy is to ensure that $|U^*(\boldsymbol{s}) - U(\boldsymbol{s})|$ is small relative to discretization errors and $U^*(\boldsymbol{s}) \to u(\boldsymbol{s})$ as $h \to 0$. This can be ensured provided $\hat{X}$ covers a neighborhood of the $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory **and** $U^* = \hat{U}$, the solution that FMM would have produced on $\hat{X}$. This is precisely what AA*-FMM does when used with an inconsistent φ; on the other hand, the different order of acceptance under SA*-FMM typically results in $U^* \neq \hat{U}$ and a lack of convergence (or a very slow convergence – see Section 2.4.1) under grid refinement.

2.3.5 The new method: AA*-FMM

The AA* technique is also quite easy to use in the continuous setting as a modification of the standard FMM. Our current implementation is based on the upwind discretization (2.6) on a standard 4-point stencil, but the required FMM-changes would be the same for any other monotone-causal stencil (either on a grid or on a simplicial mesh). Similarly

[2] The computational savings of 50% were reported in [84, 85] for $f \equiv 1$ on the domain with obstacles. Based on the above discussion, such savings are in fact highly dependent on the size of $G(\boldsymbol{s})$ relative to the total number of meshpoints. This percentage is, in turn, defined by the type of the mesh and the positions of $\boldsymbol{s}$ and $\boldsymbol{t}$ relative to the obstacles.

to a version of AA* for graphs:

- we rely on an overestimate Ψ of the time along the $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory (see section 2.3.1);
- the CONSIDERED nodes are still sorted by U-values, and thus the underestimate φ does not have to satisfy any consistency conditions;
- we only mark a node $\boldsymbol{x}$ as CONSIDERED if it satisfies the *"A* condition"*

$$U(\boldsymbol{x}) \;+\; \varphi(\boldsymbol{x}) \;\;\leq\;\; \Psi. \tag{2.14}$$

This simple criterion allows for AA* to be adapted to *both* label-setting and label-correcting methods. If φ satisfies the consistency condition (2.13), the values produced by AA*-FMM are also the same as those resulting from FMM, but on a smaller (ACCEPTED) subset of the grid $\hat{X}$. However, AA*-FMM can be also used even if φ does not satisfy (2.13), which results in additional errors but does not prevent the convergence to viscosity solution of the PDE under grid refinement.

To illustrate the efficiency of the AA*-type domain restrictions, we consider the boundaries of 3 sets:

$$\begin{aligned} C_1 &= \{\boldsymbol{x} \mid |\boldsymbol{x}-\boldsymbol{s}| + |\boldsymbol{x}-\boldsymbol{t}| \leq F_2\Psi\}, \\ C_2 &= \{\boldsymbol{x} \mid u(\boldsymbol{x}) + \varphi(\boldsymbol{x}) \leq \Psi\} \quad \subseteq\; C_1, \\ C_3 &= \{\boldsymbol{x} \mid u(\boldsymbol{x}) + v(\boldsymbol{x}) \leq \Psi\} \quad \subseteq\; C_2. \end{aligned} \tag{2.15}$$

All three are shown in Figure 2.6 for the example introduced in section 2.3.1. Both u and v are numerically approximated by FMM on the entire domain $\bar{\Omega}$. The boundaries ∂C_i are shown by bold lines for Ψ_1, Ψ_2, and Ψ_3. The set C_1 corresponds to the ellipse defined for each specific Ψ. The set $C_2 \cap L$ is roughly the set accepted by AA*-FMM with the specified Ψ **and** the underestimate φ^0. The set $C_3 \cap L$ is the minimum part of

the domain that AA*-FMM would have to accept with that Ψ even if we were to use the perfect $\varphi = \bar{\varphi}_1$. If $\Psi = \Psi_3$, then C_3 collapses to the optimal trajectory.

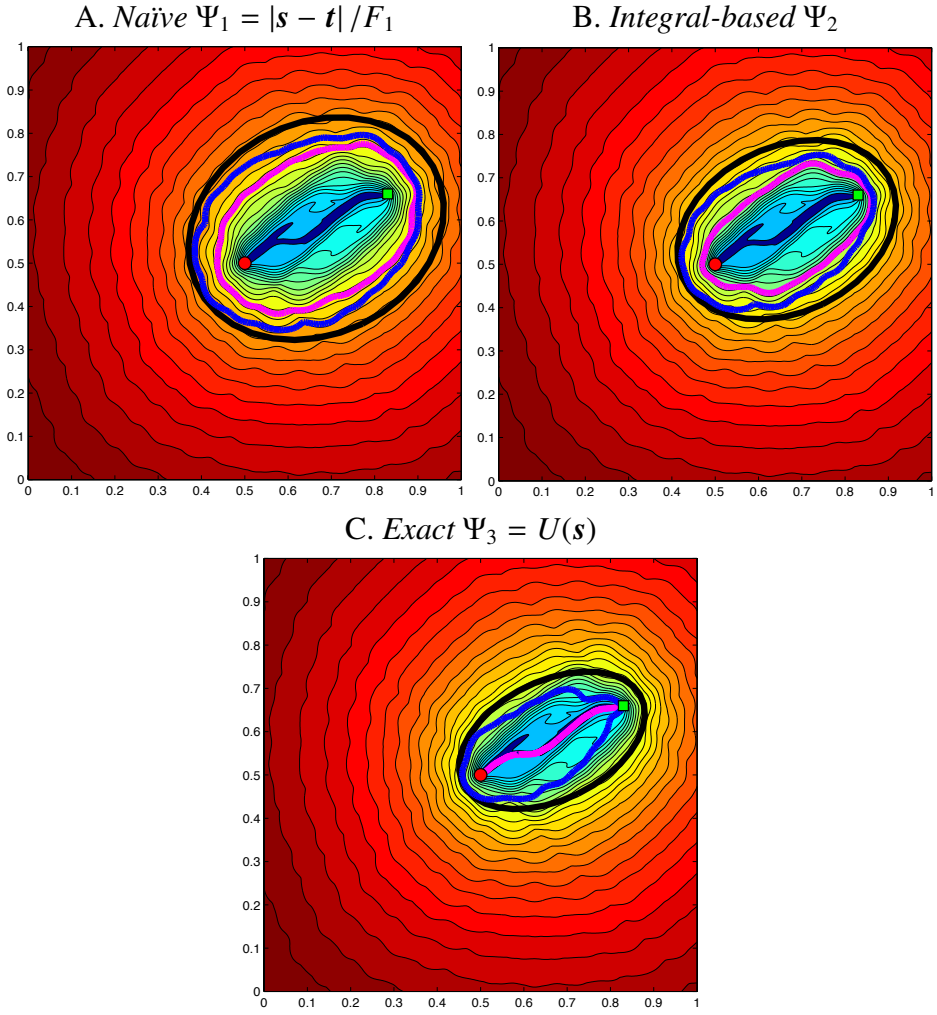


Figure 2.6: Level sets for $u + v$ computed by FMM on a 401^2 grid. In each subfigure, the bold lines show the boundaries of C_1, C_2, and C_3 (from out-to-in) for the specified Ψ.

This Figure also clearly demonstrates the importance of an accurate Ψ for the efficiency of the domain restriction in AA*-FMM. If the initial Ψ is not particularly tight, the performance can be further improved by decreasing Ψ *dynamically* in a Branch&Bound fashion. This approach relies on availability of a *"heuristic overesti-*

mate" $\psi(\boldsymbol{x}) \geq v(\boldsymbol{x}),\ \forall\, \boldsymbol{x} \in \bar{\Omega}$. For a convex domain, our implementation uses an obvious (and cheaply computed) overestimate

$$\psi(\boldsymbol{x}) \;=\; |\boldsymbol{x} - \boldsymbol{s}| \;/\; F_1, \tag{2.16}$$

which is consistent with our definition of Ψ_1 in §2.3.1. Each time a gridpoint $\boldsymbol{x}$ is ACCEPTED, this AA*BB-FMM algorithm attempts to decrease Ψ as follows:

$$\Psi \;\leftarrow\; \min\{\Psi,\ U(\boldsymbol{x}) + \psi(\boldsymbol{x})\}\,. \tag{2.17}$$

A better ψ can be obtained by numerically integrating the slowness along any feasible $(\boldsymbol{s}, \boldsymbol{t})$ trajectory, or even using PMP-based techniques. Performing such computations for every ACCEPTED gridpoint would be clearly prohibitive, but using it every so often (in addition to the systematic use of formula (2.16)) could be a useful technique to investigate in the future.

Remark 5. *On graphs, using the exact "underestimate" $\varphi = V$ simply resulted in accepting only those nodes that lie on the optimal path. In the continuous case, the optimal trajectory does not pass through every node it directly depends on. Even for a node $\boldsymbol{x}$ immediately next to the optimal $(\boldsymbol{s}, \boldsymbol{t})$ trajectory, if the underestimate φ is very accurate, this may cause the A* condition* (2.14) *to fail (resulting in $\boldsymbol{x}$ never becoming* CONSIDERED*). This situation rarely arises in practice – e.g., with $\varphi = \varphi^0$ this can happen only if Ψ is exact and the speed $f(\boldsymbol{x}) = F_2$ on some neighborhood of $\boldsymbol{s}$.*
We have used two different approaches to address this issue:

- *Introduce a numerical tolerance factor; i.e., use $(1 + \epsilon_{tol} h^{\mu})\Psi$ instead of Ψ. Our analysis of restriction-caused errors in Section 2.5 applies as long as $\epsilon_{tol} > 0$ and $\mu \in [0, 1/2)$. All the numerical tests in Section 2.4 rely on this approach and confirm the convergence even with $\mu = 1/2$.*

- *Alternatively, if $\boldsymbol{s}$ has not been accepted by the end of AA*-FMM, one can simply take $U(\boldsymbol{s}) = \Psi$. Since Ψ was obtained as a cost of some known $(\boldsymbol{s}, \boldsymbol{t})$ trajectory, that trajectory is then declared optimal (at least for the current grid resolution).*

2.4 Numerical Results

All algorithms were implemented in C++ and compiled with `g++` version 4.2 on a Macbook Pro (4 GB RAM and an Intel Core i7 processor – four 2 GHz cores). To make the benchmarking results as compiler/platform-independent as possible, we have turned off all compiler optimizations (option `-O0`). For all of the 2D and 3D examples, $\bar{\Omega} = [0, 1]^n$ is discretized by a uniform cartesian grid with m^n gridpoints. To test the numerical approximation errors in distance computations (Section 2.4.1), we have used an analytical solution $u(\boldsymbol{x}) = |\boldsymbol{x} - \boldsymbol{t}|$. In all other cases, the 'ground truth' u was computed numerically by FMM on the full domain using the 'highly' refined grid:

Dimension	**Ground truth**	**Resolutions considered**
$n = 2$	$m = 6401$	$m = 101,\ 201,\ 401,\ 801,\ 1601$ and 3201
$n = 3$	$m = 401$	$m = 26,\ 51,\ 101,\ 201$ (and 401 when $f \equiv 1$)

Accuracy metrics. Since we are interested in single-source / single-target problems, all accuracy metrics are based on comparing various numerical approximations and the true solution at a single point $\boldsymbol{s}$. As before, we use U to denote the solution produced by FMM on the entire X while U^* denotes the solutions produced by the respective A*-modifications of FMM. We base our comparison on the following "relative errors" for

each example:

$$\begin{aligned}
\mathcal{E}^d &= \text{relative discretization error (DE) at } \boldsymbol{s} \text{ using FMM} &&= |U(\boldsymbol{s}) - u(\boldsymbol{s})| \ / \ u(\boldsymbol{s}) \\
\mathcal{E}^* &= \text{relative error at } \boldsymbol{s} \text{ when using A*} &&= |U^*(\boldsymbol{s}) - u(\boldsymbol{s})| \ / \ u(\boldsymbol{s}) \\
\mathcal{E}^*_N &= \text{relative error at } \boldsymbol{s} \text{ explicitly due to A*} &&= [U^*(\boldsymbol{s}) - U(\boldsymbol{s})] \ / \ U(\boldsymbol{s}) \ \geq \ 0.
\end{aligned}$$

Since the upwind discretization is convergent, $U \to u$ and thus $\mathcal{E}^d \to 0$ as $h \to 0$. Correspondingly, a successful domain restriction should have $\mathcal{E}^* \to 0$ (and thus $\mathcal{E}^*_N \to 0$) as $h \to 0$.

To measure the efficiency of the domain restriction, we also define

$$\mathcal{P} \ = \ \text{fraction of domain computed} \ = \ (\#\text{ of gridpoints ACCEPTED or CONSIDERED}) \ / \ m^n.$$

Underestimate functions. In all examples except for section 2.4.4, we rely on naïve and scaled-oracle heuristics (i.e., φ^0_λ and $\bar{\varphi}_\lambda$). We consider this sufficient since the accuracy of the AA* approach is really underestimate-neutral (though the efficiency is clearly dependent on both φ and Ψ). We expect that the results based on any other heuristics (including those in [57, 58, 59]) will be qualitatively similar.

2.4.1 Constant speed $f \equiv 1$ in 2D and 3D

In the constant speed case, all characteristics are straight lines and the the naïve heuristic coincides with the actual time-to-go (i.e., $\varphi^0 = v$). In this subsection we use the underestimate $\varphi = \varphi^0_\lambda$ and place $\boldsymbol{s}$ and $\boldsymbol{t}$ at opposite corners of $\bar{\Omega}$. Our goal is to test the effect of $\lambda \in [0, 1]$ on the accuracy and efficiency for different grid resolutions $h = 1/(m-1)$. In testing AA*-FMM, we use $\Psi = (1 + \epsilon_{tol} h^\mu)|\boldsymbol{s} - \boldsymbol{t}|$, where $\mu = 1/2$ with $\epsilon_{tol} = 1/4$ in 2D and $\epsilon_{tol} = 1/3$ in 3D. This ensures that AA*-FMM does not terminate before $\boldsymbol{s}$ is ACCEPTED and also results in the set $C_3 = C_2$ shrinking to a straight line as $h \to 0$.

Figure 2.7 shows the level sets of U^* computed by SA*-FMM and AA*-FMM on a 2D grid with $m = 351$ and $\lambda \in \{0.25,\ 0.5,\ 0.75, 1\}$. The non-smoothness of the level-sets produced by SA*-FMM is due to the additional errors introduced by that method. For $\lambda = 1$ these errors also result in a larger $\mathcal{P}$ – despite the fact that our AA*-FMM has a built-in "restriction slackness" (since $\Psi > u(\boldsymbol{s})$). Figure 2.8 shows $\log_{10}(\mathcal{E}_N^*)$ as m and λ vary. For $\lambda \geq 0.55$, the errors produced by SA*-FMM are not only relatively large, but also do not decrease much under grid refinement. In contrast, the errors in AA*-FMM decrease quite rapidly even though the set C_3 is also shrinking as $h \to 0$; see also the convergence analysis in Section 2.4.

Since in this example $G(\boldsymbol{s}) = X$, additional errors should result from *any* domain restriction. However, the finite-precision of the floating point arithmetic results in "zero domain restriction errors" (white spaces in Figure 2.8) for AA*-FMM even for many test runs where $G(\boldsymbol{s})$ is partly truncated. E.g., see the case ($\lambda = 0.75$, $m = 351$) in Figures 2.7 and 2.8.

Figures 2.9 and 2.10 show the full accuracy/efficiency data holding $\lambda = 1$ and varying m.

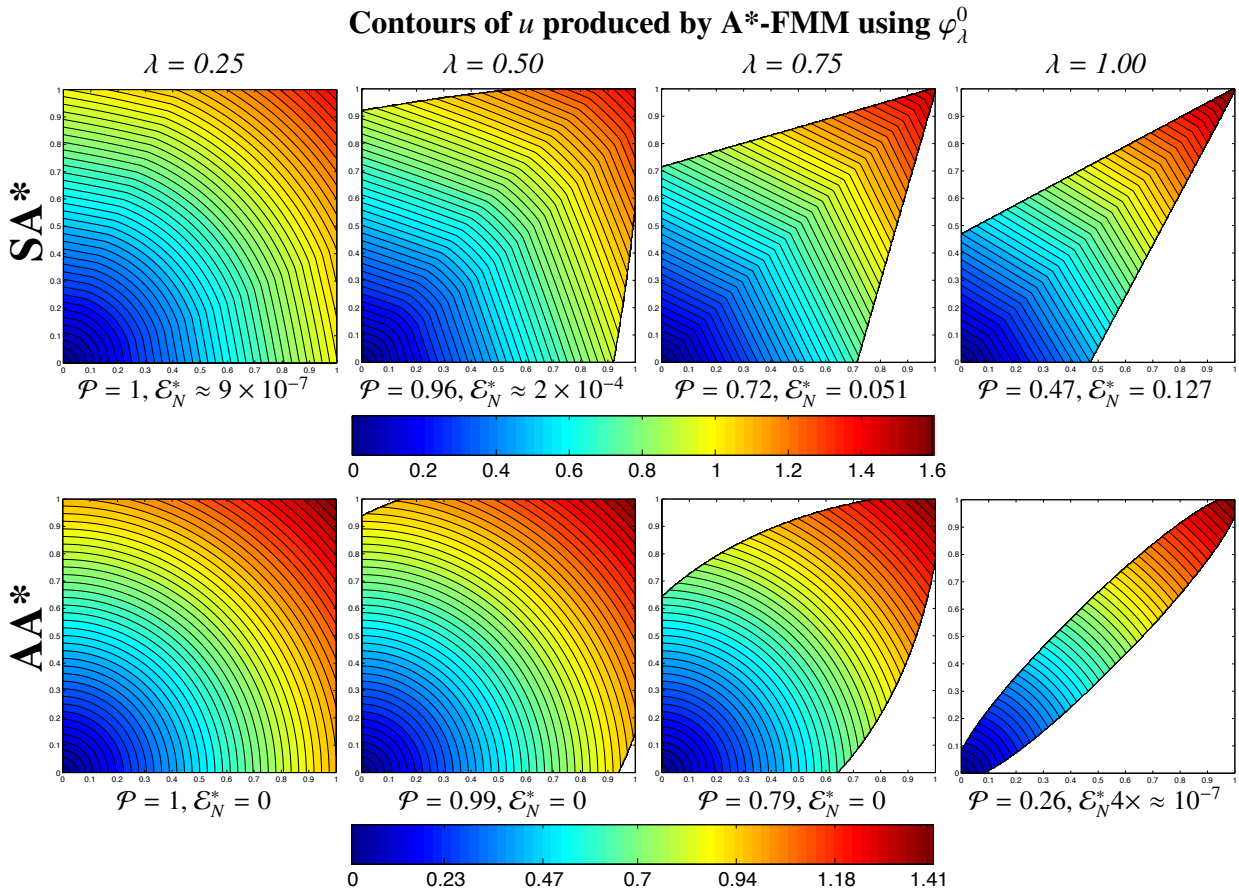


Figure 2.7: The top row was produced with SA*-FMM and the error can be seen in two ways: (1) the deformation of the level sets and (2) the value at the source is ≈ 1.61. The bottom row shows the results of AA*-FMM. We hold $m = 351$ while λ values increase from left to right.

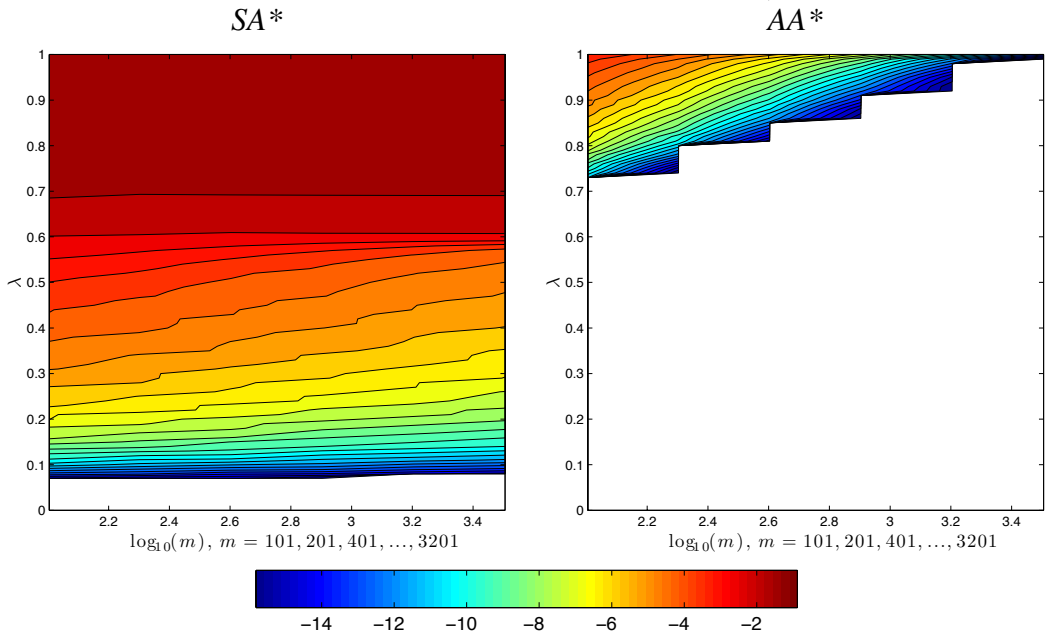


Figure 2.8: SA* versus AA* comparison based on $\mathcal{E}_N^*$ errors. The horizontal axis shows the grid resolution m, and the vertical axis corresponds to the heuristic strength λ. White corresponds to errors smaller than the machine ε.

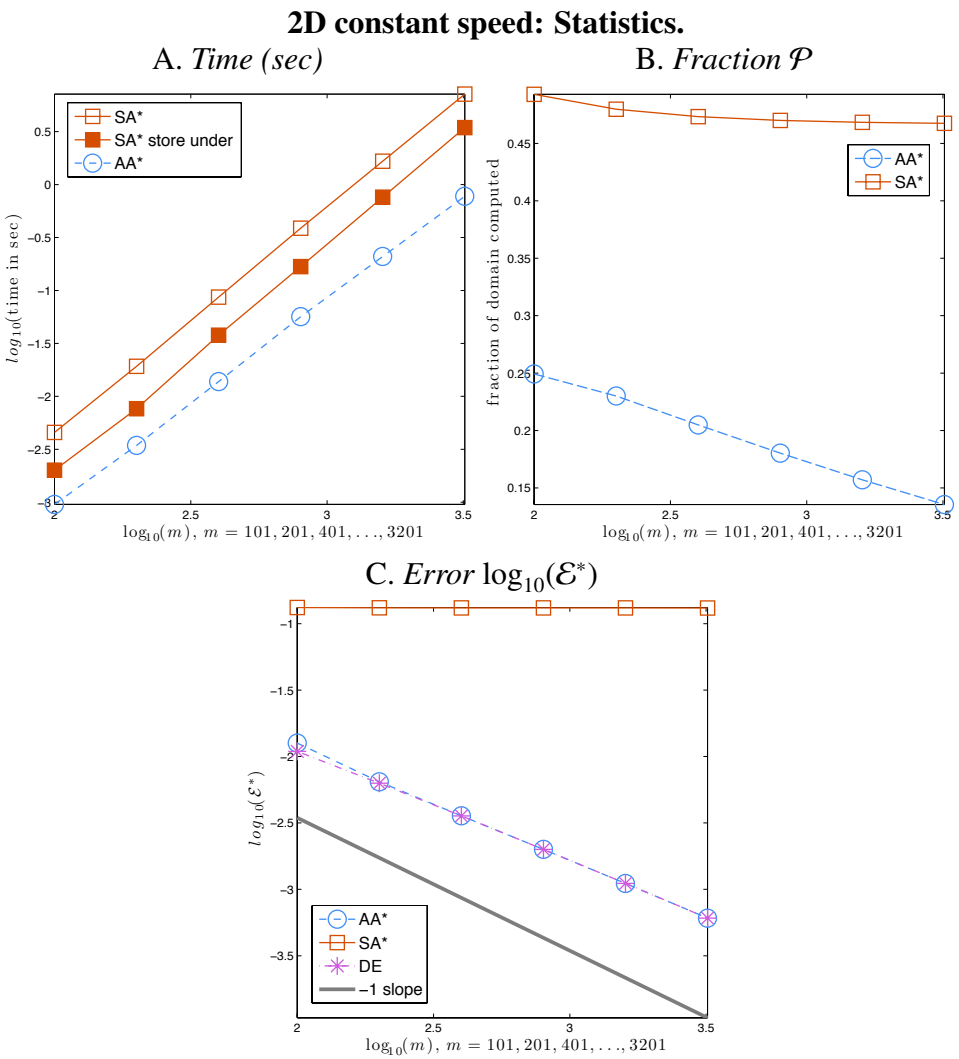


Figure 2.9: The CPU-time, the fraction $\mathcal{P}$ of the domain computed, and the error $\mathcal{E}^*$ for both SA* and AA* using a constant speed function in 2D. The solid square markers in the time plot indicate the time for a version of SA*-FMM that stores each $\varphi(\boldsymbol{x})$ after it is first computed. The underestimate function used is φ^0 and The benchmarking is performed for $\lambda = 1$ (i.e., corresponding to the very top slice in Figure 2.8).

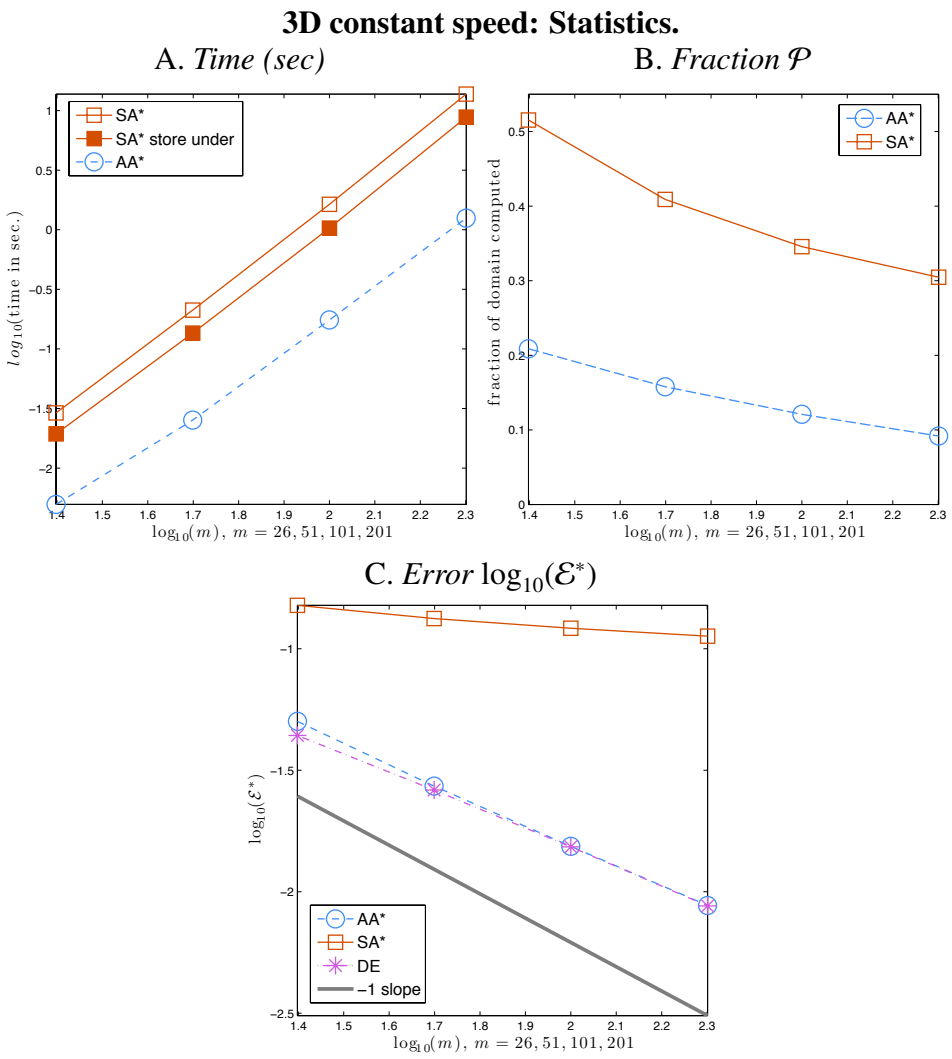


Figure 2.10: The same data as in Figure 2.9, but for 3D computations.

2.4.2 Oscillatory speed function in 2D and 3D

For the next 2D example, we set $\boldsymbol{s} = (0.95, 0.7)$ and $\boldsymbol{t} = (0.5, 0.5)$ and consider a highly oscillatory speed

$$f(x, y) \quad = \quad 1 \; + \; 0.5 \sin(20\pi x) \sin(20\pi y), \tag{2.18}$$

resulting in frequent directional changes along most optimal paths. We start by focusing on a scaled oracle heuristic $\varphi = \bar{\varphi}_\lambda$ with AA*-FMM also relying on $\Psi = (1 + \epsilon_{tol} h^\mu) v(\boldsymbol{s})$. Figure 2.11 shows the level sets of numerical solutions obtained with $m = 401$. We note that the SA*-errors result in a significant distortion of the optimal trajectory (see the switch between $\lambda = 0.3$ and $\lambda = 0.7$).

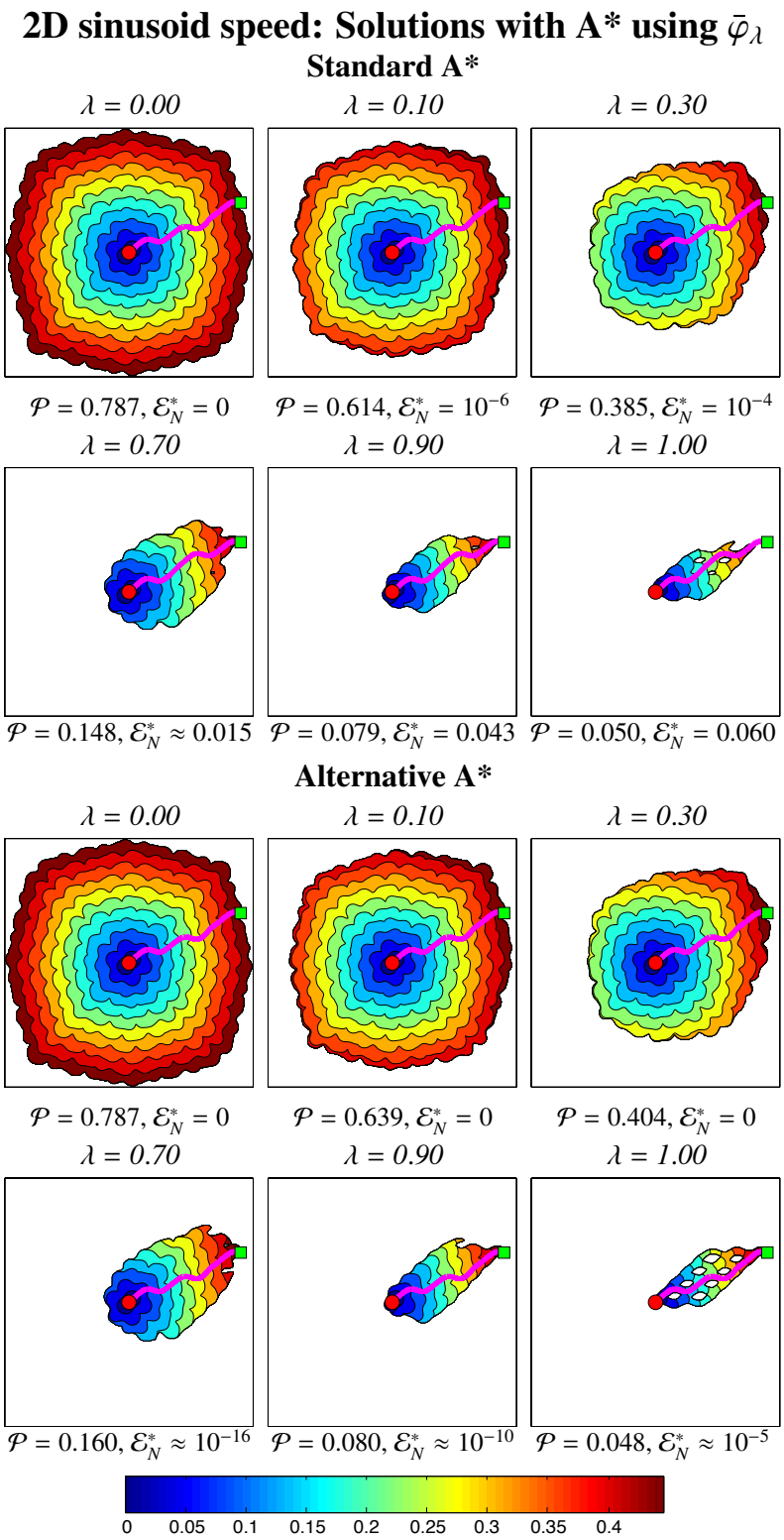


Figure 2.11: Numerical results of FMM combined with SA* and AA*, showing the fraction of domain computed $\mathcal{P}$ and the relative error $\mathcal{E}_N^*$. Note the change in the "optimal" trajectory for SA* between $\lambda = 0.3$ and $\lambda = 0.70$. The solutions were produced using $m = 401$.

Figure 2.12 compares the accuracy of these techniques for different (m, λ) pairs. Qualitatively the picture is largely the same as in Figure 2.8, but with two non-trivial differences. First, the 'white block' in the lower-left corner of the SA* plot indicates the lack of additional errors with $m = 101$ and $\lambda \leq 0.15$. Based on our computational experiments, this is an *extremely* rare situation – the only example we could find, where the entire $G(\boldsymbol{s})$ is processed by SA*-FMM in the correct order despite the fact that the heuristic φ is inconsistent. Second, we observe that the AA*-FMM-generated errors are not always monotone decreasing in m. E.g., the errors are present for $(m = 401,\ \lambda = 0.75)$, but not for $(m = 201,\ \lambda = 0.75)$, where the entire $G(\boldsymbol{s})$ is accepted.

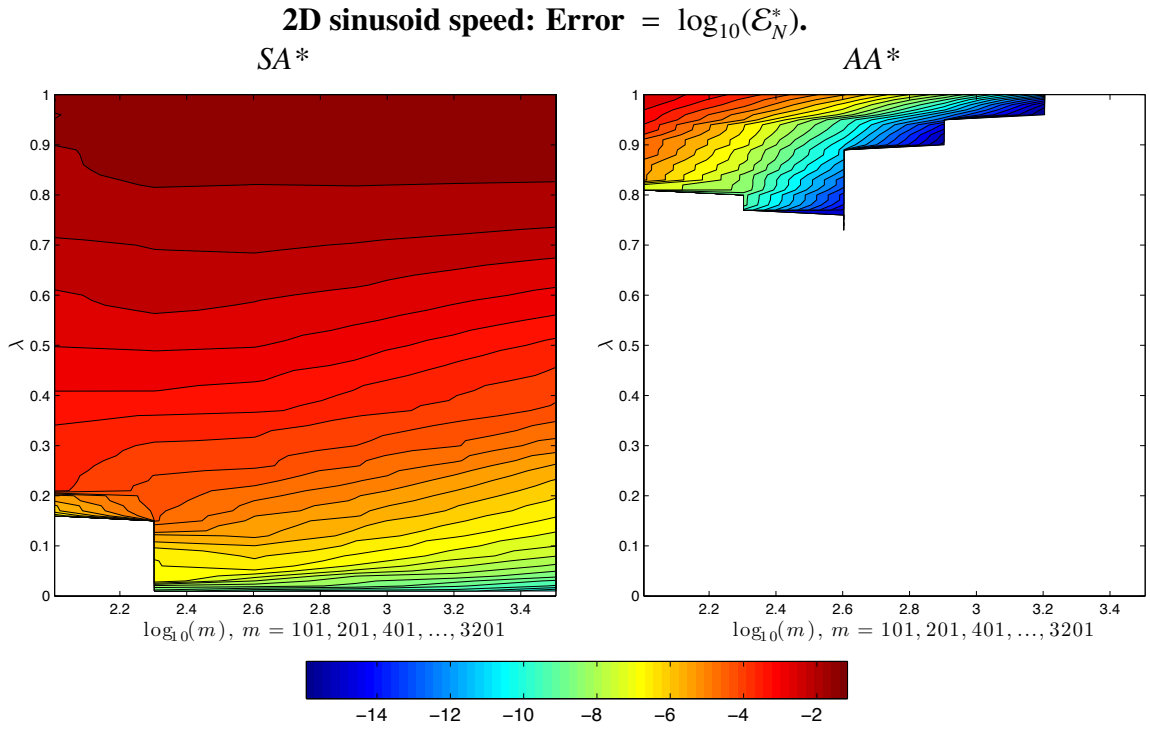


Figure 2.12: This plot shows the same results as Figure 2.8 except with the sinusoid speed (2.18).

Since the oracle heuristic is generally unavailable, we now consider the accuracy/efficiency tradeoffs using the naïve heuristic $\varphi = \varphi^0$ and a realistically obtainable (but conservative) overestimate $\Psi = \Psi_2$. Figure 2.13 shows that AA*-FMM yields com-

parable efficiency (despite accepting a larger part of the domain) while also ensuring $U^*(\boldsymbol{s}) = U(\boldsymbol{s})$ since the entire $G(\boldsymbol{s})$ is accepted.

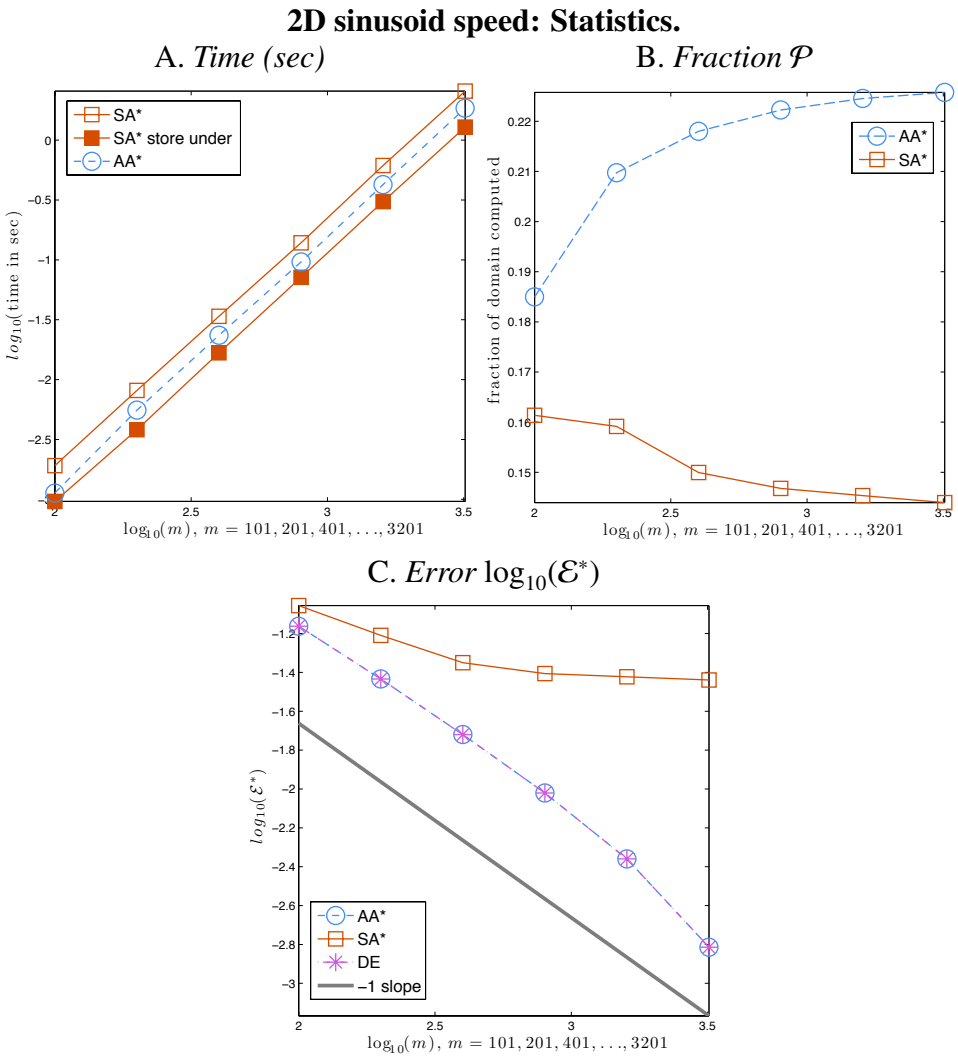


Figure 2.13: These results show the time (in seconds), fraction domain calculated, and the error $\mathcal{E}^*$ for both SA* and AA* using a highly oscillatory sinusoid function in 2D. The naïve heuristic was used, and for AA* $\Psi = \Psi_2$.

We also consider similar oscillatory examples in 3D with

$$f(x, y, z) = 1 + A \sin(10\pi x) \sin(10\pi y) \sin(10\pi z), \tag{2.19}$$

for two amplitudes $A = 0.1$ and $A = 0.35$. The source/target locations are $\boldsymbol{s} =$

$(0.72, 0.6, 0.8)$ and $\boldsymbol{t} = (0.32, 0.4, 0.36)$. Figure 2.14 shows the accuracy/efficiency data based on realistic $\varphi = \varphi^0$ and $\Psi = \Psi_2$. The errors due to AA* are negligible compared to discretization errors, while the errors due to SA* are again quite noticeable and decrease much slower as $h \to 0$.

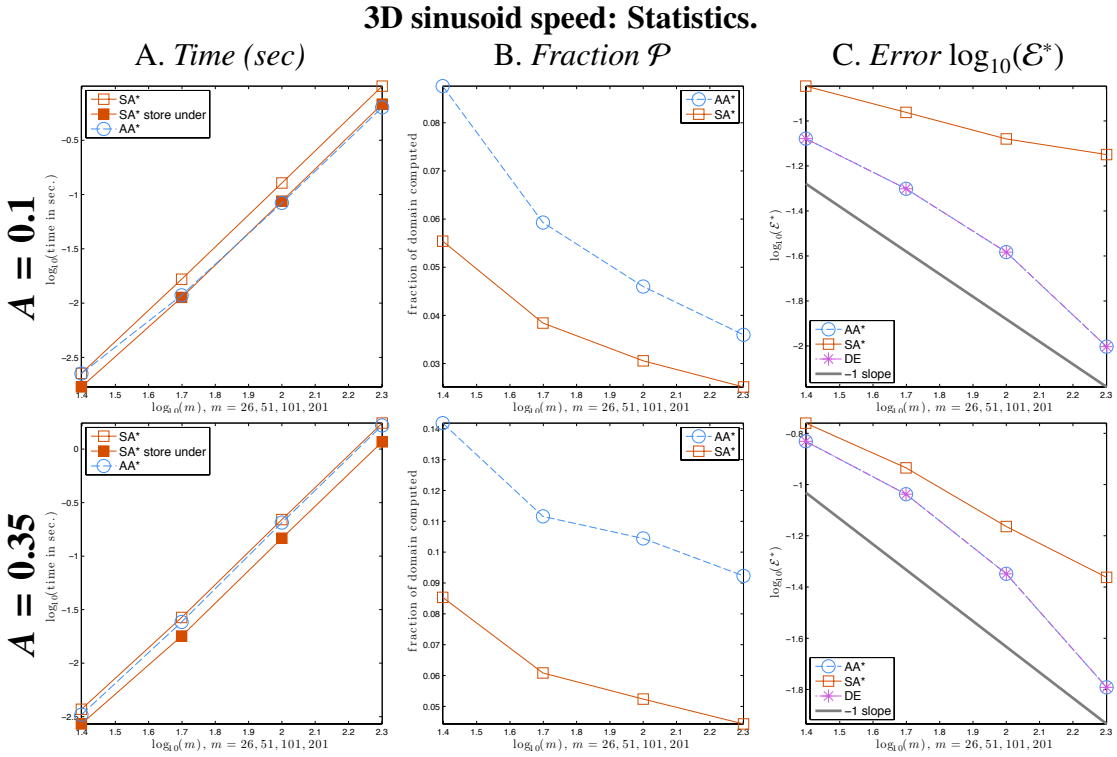


Figure 2.14: These results again show the average time (in seconds) of 10 trial runs, fraction domain calculated, and the error $\mathcal{E}^*$ for both SA* and AA* using (2.19). The naïve heuristic was used, and for AA* $\Psi = \Psi_2$. The top row corresponds to $A = 0.1$ and the bottom row shows the results when $A = 0.35$. When $A = 0.1$ the result might seem counterintuitive: AA* takes less CPU time even though it processes more of the domain. Careful profiling shows that for SA* the three-neighbor update fails more frequently and causes the algorithm to perform more two-sided updates. This makes an average node update in SA* more computationally expensive; hence the slower time.

2.4.3 Satellite image

The following path-planning example is borrowed from [57, 58, 59]. The grayscale intensities of a satellite photograph (Figure 2.15A) are imported into the range $[0, 755]$ using MATLAB's `imread()` routine. For a given gridpoint $\boldsymbol{x}$ assume that it falls into a pixel with grayscale value $i(\boldsymbol{x}) \in [0, 755]$. This then defines the speed $f : \bar{\Omega} \to [0.001, 1.001]$ via rescaling:

$$f(\boldsymbol{x}) \;=\; 0.001 \;+\; i(\boldsymbol{x})/755$$

This is the same intensity/speed mapping used in [59], but our experimental setup is slightly different:

- Unlike Peyré et al., we omit the pre-smoothing of the original 744×744 image and simply downsample it to 350×350.
- Peyré et al. use $\varphi = \varphi^C_{\lambda,R}$; they fix $\lambda = \frac{1}{2}$ and vary R. Instead, we first use $\varphi = \varphi^0$ (Figure 2.15B) and then switch to $\varphi = \bar{\varphi}_\lambda$ (Figure 2.16). Unlike with $\varphi^C_{\lambda,R}$, the use of $\bar{\varphi}_\lambda$ directly illustrates the performance of A* as the quality of φ improves.
- We use slightly different source and target locations ($\boldsymbol{s} = (337h, 161h)$ and $\boldsymbol{t} = (16h, 188h)$; see Figure 2.15C). Our $\boldsymbol{s}$ falls on the opposite side of a shockline compared to $\boldsymbol{s}$ used in [59].

Figure 2.15B shows the level sets of the solution on the full domain, with ∂L and ∂C_2 (using φ^0 and Ψ_3) shown in bold. (The set ACCEPTED by SA* is approximately the same as AA*.)

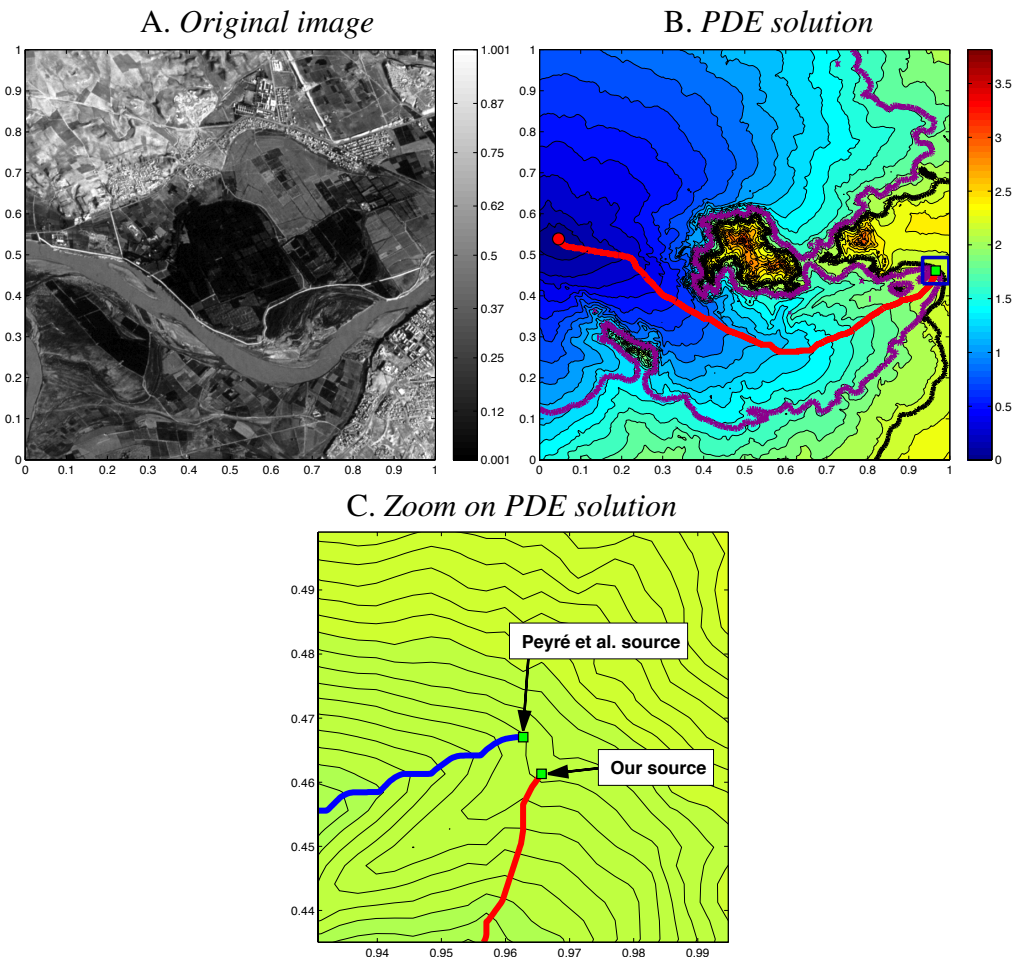


Figure 2.15: A. The original satellite image mapped to a speed $f \in [0.001, 1.001]$. B. The solution to the PDE on a 350×350 grid with ∂L and ∂C_2 (using φ^0 and Ψ_3) drawn in bold. C. The upper marker is approximately the same $\boldsymbol{s}$ as in [59]; the lower marker is the same $\boldsymbol{s}$ used in B and Figure 2.16A.

This example illustrates the use of A*-techniques with a discontinuous speed function. The rather limited computational savings in 2.15B are clearly caused by the use of an "overly optimistic" φ^0. However, the lack of accuracy of this naive underestimate is not caused by any discontinuities in f – instead it is simply a result of a large F_2/F_1, with f values much closer to F_1 on most of the domain.

We now switch to "oracle tests" with $\varphi = \bar{\varphi}_\lambda$ and AA*-FMM relying on $\Psi = \left(1 + \sqrt{h}/8\right) V(\boldsymbol{t})$. Using this heuristic, the domain restriction becomes much more effective for both SA* and AA*. But since $\boldsymbol{s}$ is close to a shockline, additional errors due to SA*-FMM are sufficiently large to change the optimal trajectory in several ways (see Figure 2.16A). In contrast, the errors from AA*-FMM are much smaller and the optimal trajectory remains the same for all λ.

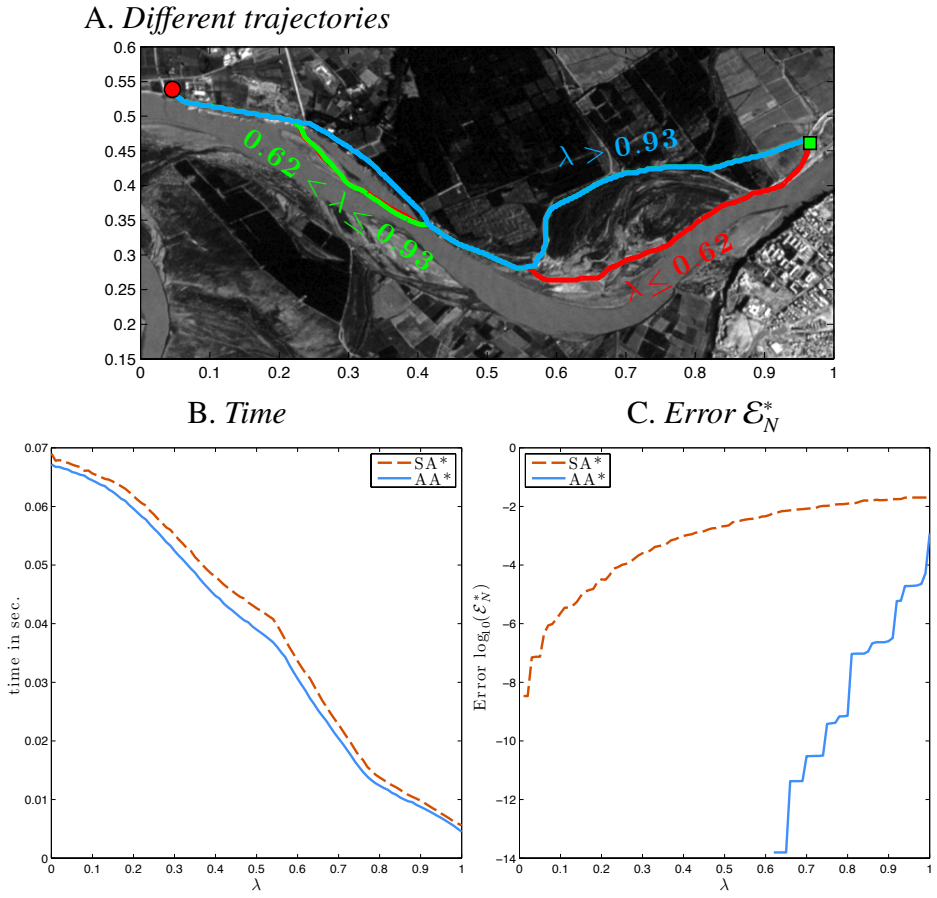


Figure 2.16: A. The λ-dependent "optimal" trajectories recovered by SA*-FMM (red, green, and blue curves). AA*-FMM always recovers the truly optimal (red) trajectory. When the trajectories overlap, the red red curve lies under the green, and the green curve lies under the blue. B&C. The time (in seconds) and $\mathcal{E}_N^*$ produced by SA* and AA* as λ changes in [0, 1].

2.4.4 Replanning in a dynamic environment

Our final example illustrates several important points:

1. The use of special/custom underestimate φ based on a related control problem.
2. The optimal trajectory from a related control problem is valuable as an initial guess for the Pontryagin Maximum Principle (PMP).
3. The PMP-computed trajectory is not necessarily globally optimal, but can be used to produce an accurate Ψ.

Here we will use a slightly more general setup where the task is to minimize the total *cost* (instead of considering only the *time*) to reach $\boldsymbol{t}$. Given a running cost function $K : \Omega \to (0, +\infty)$ integrated along the trajectory and a speed f_0, the value function u now satisfies a different Eikonal PDE given by

$$|\nabla u(\boldsymbol{x})|\, f_0(\boldsymbol{x}) \;=\; K(\boldsymbol{x}). \tag{2.20}$$

Our specific problem is to find the "safest" trajectory in an adversarial environment, with K higher on the parts of the domain more closely monitored by the adversary.

If we assume no prior information on enemy locations and monitoring patterns, it is natural to select $K \equiv 1$, which implies that the quickest trajectory is in fact the safest. Consider the domain $\bar{\Omega} = [-0.05, 0.85]\times[0, 0.9]$ with the speed and running-cost defined by

$$f_0(x, y) \;=\; 1 \;+\; 0.99 \sin(4\pi x) \sin(4\pi y) \quad \text{and} \quad K_0 \equiv 1.$$

The solution u to this *"no enemy observers"* problem is shown in Figure 2.17C. Figure 2.17A shows the contours of f_0 with two locally optimal trajectories. The 'upper' solid trajectory is globally optimal and found by tracing the gradient of u; the 'lower' locally optimal trajectory is computed using PMP.

Our perception of the trajectory safety will change once we discover specific locations of enemy observers. For example, if we know that there are two observers located at $\boldsymbol{x}_1 = (0.50, 0.77)$ and $\boldsymbol{x}_2 = (0.33, 0.45)$, we might encode this new information in the cost function:

$$K(\boldsymbol{x}) \;=\; 1 \;+\; 2\exp\left(\frac{|\boldsymbol{x}-\boldsymbol{x}_1|^2}{0.01}\right) \;+\; 8\exp\left(\frac{|\boldsymbol{x}-\boldsymbol{x}_2|^2}{0.002}\right).$$

The solution to (2.20) with speed f_0 and the above cost K can be shown to satisfy (3.1) with $f = f_0/K$. The contours of this new modified speed f can be seen in Figure 2.17B with two "locally safest" trajectories that can be viewed as perturbations of the locally time-optimal paths from Figure 2.17A. Note that, because of the higher cost around $\boldsymbol{x}_1$, the 'upper' locally optimal trajectory is no longer globally optimal.

The full solution to the time-optimal problem becomes useful if we want to introduce A* techniques for all *"multiple enemy observers"* problems. Let $V_0(\boldsymbol{x})$ be the minimum time to reach $\boldsymbol{s}$ using the speed f_0. Suppose that V_0 is pre-computed by FMM and stored for the entire X. Returning to the problem with known observers, we may take $\varphi = V_0$ since $f_0 \le f \implies V_0 \le V$. The overestimate Ψ can be obtained by integrating K/f_0 along any feasible trajectory. If we use the globally time-optimal trajectory (the solid black curve in Figure 2.17A), this yields a good $\Psi_A \approx 0.6752$. An even better overestimate $\Psi_B \approx 0.6447$ is obtained if we use the globally time-optimal trajectory as the initial guess for PMP, and then integrate K/f_0 along the resulting "locally safest" trajectory (the 'upper' black curve in Figure 2.17B).

Figure 2.17D shows the solution level sets for the *"multiple enemy observers"* problem together with boundaries of several computational sets. The bold black curve is ∂L, showing the part of $\bar{\Omega}$ ACCEPTED by FMM. The next (inward) bold curve is ∂C_2 with $\Psi = \Psi_B$ – the boundary of a subset ACCEPTED by AA*-FMM. The final bold curve is ∂C_2 with $\Psi = U(\boldsymbol{s})$ – this approximates the boundary of a subset ACCEPTED by SA*-FMM.

Method	**Time (seconds)**	**Ratio $\mathcal{P}$**	**Error $\mathcal{E}^*_{\mathbf{N}}$**
FMM	0.02665	0.82	0
SA-FMM*	0.004950	0.130	0.02550
AA-FMM,* $\Psi = \Psi_B$	0.0101	0.29	1.77×10^{-11}
AA-FMM,* $\Psi = U(\boldsymbol{s})$	0.00526	0.14	0.00150

Table 2.1: Statistics of the algorithms used in Section 2.4.4.

(AA*-FMM would also restrict to the latter set, but only if we were lucky enough to start with Ψ corresponding to the 'lower' curve in Figure 2.17B).

Even though AA*-FMM computes the solution on a larger part of the domain, its computational efficiency is still comparable and the accuracy is superior to SA*-FMM. For example, with $m = 201$ (using 100 trial runs averaged for the time) we report the timings in Table 2.1.

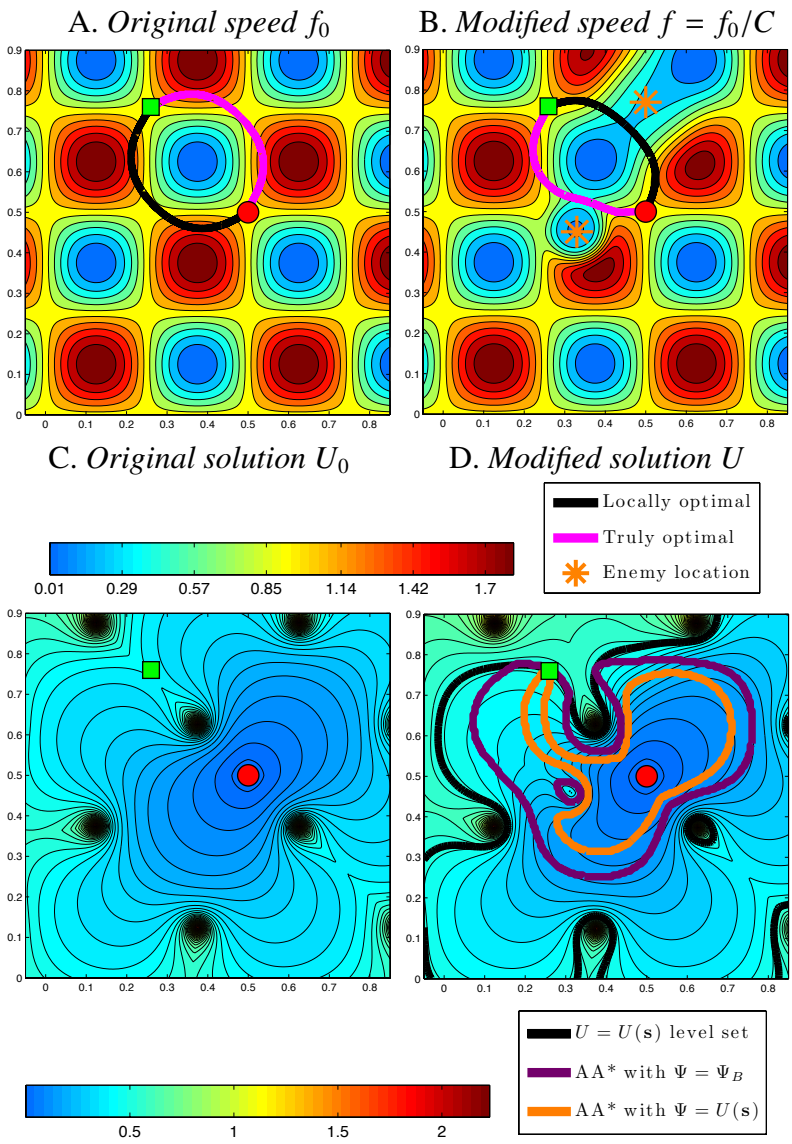


Figure 2.17: **A.** Contours of the original speed function f_0. **B.** Contours of "*modified speed function*" $f = f_0/K$ with the enemy locations shown by asterisks. **C.** Contours of the original solution to the problem with speed f_0 and constant running cost. **D.** Contours of the solution corresponding to the modified speed function $f = f_0/K$ with ∂L drawn in bold black. ∂C_2 is in dark purple using $\Psi = \Psi_B$, and in orange when using $\Psi = U(\boldsymbol{s})$.

2.5 Justification of AA* Convergence

If AA*-FMM is used with an inconsistent heuristic φ, the domain restriction usually affects the dependency graph (i.e., $G(\boldsymbol{s}) \not\subset \hat{X}$), and the produced solution is larger than would result from running FMM on the full grid: $U^*(\boldsymbol{s}) > U(\boldsymbol{s})$. In this section we analyze why $(U^*(\boldsymbol{s}) - U(\boldsymbol{s})) \to 0$ as $h \to 0$.

We first note that the answer is simple if there exists an open set $\Omega_0 \subset \Omega$ such that

- the $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory lies in Ω_0, and
- and all gridpoints falling into Ω_0 are accepted by AA*-FMM *regardless of h*.

In this case, an $\bar{\Omega}_0$-constrained viscosity solution will already yield the correct $u(\boldsymbol{s})$ in the limit. In previous sections, we showed that such Ω_0 often arises because Ψ and/or φ are not tight. But if the over/underestimates also improve in quality as $h \to 0$, then the AA*-FMM accepted region shrinks under grid refinement, and a more careful argument is needed to explain the convergence.

To address this, we compare solutions produced by the original FMM solving the same discretized system (2.6) but on different grid subsets and with different boundary conditions. For the rest of this section, we will not rely on the fact that $\hat{X}$ is defined through AA*-FMM. As a benefit, our error analysis is also relevant for domain decomposition-based parallelizations of FMM; e.g., see [13].

Consider a restriction of FMM computations to any $\hat{X} \subset X$ containing both $\boldsymbol{s}$ and $\boldsymbol{t}$, and define the "restriction boundary" set $\Xi = \{\boldsymbol{x} \in X \backslash \hat{X} \mid N(\boldsymbol{x}) \cap \hat{X} \neq \emptyset\}$. For notational simplicity, we will assume that the $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory is unique and that the upwind neighbors $(\boldsymbol{x}_H, \boldsymbol{x}_V)$ are uniquely defined for every gridpoint $\boldsymbol{x}$.

We will discuss the relationship between the following discretized solutions:

- As before, U denotes the solution on the entire X with the boundary condition $U(\boldsymbol{t}) = 0$.
- $\hat{U}$ denotes the solution on $\hat{X}$ with the same boundary condition $\hat{U}(\boldsymbol{t}) = 0$. We can also interpret it as a solution on $\hat{X} \bigcup \Xi$ with $Q = \{\boldsymbol{t}\} \bigcup \Xi$ and $q = +\infty$ on Ξ. Recall that, if $\hat{X}$ is defined as the set of nodes ACCEPTED by AA*-FMM, then this method also produces the same solution (i.e., $U^* = \hat{U}$ on $\hat{X}$).
- $\bar{U}$ denotes the solution computed on $\hat{X} \bigcup \Xi$ with $\bar{U}(\boldsymbol{t}) = 0$ and the more general boundary conditions $\bar{U}(\boldsymbol{x}_i) = q_i$ specified $\forall\, \boldsymbol{x}_i \in \Xi$.

Observation 1. *The following properties are easy to verify based on the causality of* (2.6)*:*

1. $q_i = U_i,\ \forall\, \boldsymbol{x}_i \in \Xi \quad \Longrightarrow \quad \bar{U}_j = U_j,\ \forall\, \boldsymbol{x}_j \in \hat{X}$;
2. $q_i \geq U_i,\ \forall\, \boldsymbol{x}_i \in \Xi \quad \Longrightarrow \quad \bar{U}_j \geq U_j,\ \forall\, \boldsymbol{x}_j \in \hat{X}$;
3. $\hat{U}_j \geq \bar{U}_j,\ \forall\, \boldsymbol{x}_j \in \hat{X}$;
4. *Suppose C is a constant such that $C \geq \max_{\boldsymbol{x}_j \in \hat{X}} \hat{U}_j$. Then*
   $q_i \geq C,\ \forall\, \boldsymbol{x}_i \in \Xi \quad \Longrightarrow \quad \bar{U}_j = \hat{U}_j,\ \forall\, \boldsymbol{x}_j \in \hat{X}$.
5. *Suppose $D(\boldsymbol{x})$ is the arclength of the shortest grid-aligned path within $\hat{X}$ from $\boldsymbol{x}$ to $\boldsymbol{t}$. Then $C = \max_{\boldsymbol{x}_j \in \hat{X}} D(\boldsymbol{x})/F_1 \geq \max_{\boldsymbol{x}_j \in \hat{X}} \hat{U}_j$.*

For any specific $\boldsymbol{x}_i \in X$, if we define $\hat{X} = X \backslash \{\boldsymbol{x}_i\}$ and choose $q_i > U_i$ this might result in $\hat{U}(\boldsymbol{s}) > U(\boldsymbol{s})$. This "add-one-gridpoint-to-Q" procedure motivates our definition of *sensitivity coefficients*:

$\alpha_i = \alpha(\boldsymbol{x}_i) = \dfrac{\partial U(\boldsymbol{s})}{\partial U_i}$ or, more rigorously, $\alpha_i = \dfrac{\partial \hat{U}(\boldsymbol{s})}{\partial q_i}$ computed on $\hat{X} = X \backslash \{\boldsymbol{x}_i\}$ with $q_i = U_i$.

Due to the monotonicity of (2.6), $\alpha_i \geq 0$ and it is strictly positive if and only if $\boldsymbol{x}_i \in G(\boldsymbol{s})$.

Lemma 2. *The net effect of a domain restriction can be bounded from above using α's even for a* general *set $\hat{X}$:*

1. *If $q(\boldsymbol{x}) \geq U(\boldsymbol{x})$, $\forall\, \boldsymbol{x} \in \Xi$, then $\bar{U}(\boldsymbol{s}) - U(\boldsymbol{s}) \;\leq\; \sum_{\boldsymbol{x}\in\Xi} \alpha(\boldsymbol{x})\,(q(\boldsymbol{x}) - U(\boldsymbol{x}))\,.$*
2. *If $C \geq \hat{U}(\boldsymbol{x})$, $\forall\, \boldsymbol{x} \in \hat{X}$, then $\hat{U}(\boldsymbol{s}) - U(\boldsymbol{s}) \;\leq\; C \sum_{\boldsymbol{x}\in\Xi} \alpha(\boldsymbol{x})$.*

Proof. The upwind finite difference discretization (2.7) is equivalent to a semi-Lagrangian discretization:

$$U(\boldsymbol{x}_{ij}) \;=\; \min_{\beta\in[0,1]} \left\{ \frac{\left|\beta \boldsymbol{x}_H + (1-\beta)\boldsymbol{x}_V - \boldsymbol{x}_{ij}\right|}{f(\boldsymbol{x})} \;+\; \beta U(\boldsymbol{x}_H) \;+\; (1-\beta) U(\boldsymbol{x}_V) \right\}. \tag{2.21}$$

Despite the very different Eulerian perspective and notation, (2.7) can be actually derived from Kuhn-Tucker optimality conditions for (2.21); see [81, 71, 83]. Moreover, the latter can be also viewed as the dynamic programming equation for a *Stochastic Shortest Path Problem* on the grid X; see [83] for a detailed discussion. In this interpretation, the transition from $\boldsymbol{x}_{ij}$ to the neighboring node (either $\boldsymbol{x}_H$ or $\boldsymbol{x}_V$) happens probabilistically, with respective probabilities β and $(1-\beta)$, and $\frac{|(\beta \boldsymbol{x}_H + (1-\beta)\boldsymbol{x}_V - \boldsymbol{x}_{ij}|}{f(\boldsymbol{x})}$ is the cost we incur for choosing this probability distribution. The process continues until we reach $\boldsymbol{t}$, and the goal is to select $\beta_* : X \to [0,1]$ that minimizes the expected cumulative cost up to that termination. We note that, for $\boldsymbol{x}_{ij} = \boldsymbol{s}$, we have $\alpha(\boldsymbol{x}_V) = (1 - \beta_*(\boldsymbol{s}))$, $\alpha(\boldsymbol{x}_H) = \beta_*(\boldsymbol{s})$ and α values on the rest of $G(\boldsymbol{s})$ can be similarly computed using (2.21) recursively; see [16]. Moreover, if we start from $\boldsymbol{s}$ and use the optimal "stochastic routing policy" $\beta_*(\cdot)$, then $\alpha(\boldsymbol{x})$ can be naturally interpreted as a probability of passing through $\boldsymbol{x}$ before arriving at $\boldsymbol{t}$.

Suppose now we use $\beta_*(\cdot)$, but on a $\hat{X}$-restricted problem, starting from $\boldsymbol{s}$ and terminating the process (+ paying the additional cost of $q(\boldsymbol{x})$) if we transition into any $\boldsymbol{x} \in \Xi$ before reaching $\boldsymbol{t}$. Denote by $\tilde{U}$ the expected total cost of using this policy and by $\tilde{\alpha}(\boldsymbol{x})$ the probability of reaching $\boldsymbol{x}$ before termination. We first note that

$\tilde{\alpha}(\boldsymbol{x}) \leq \alpha(\boldsymbol{x})$, $\forall\, \boldsymbol{x} \in \hat{X} \cup \Xi$ since some stochastic paths previously leading through $\boldsymbol{x}$ are now removed due to an earlier entry to Ξ. Secondly, $\tilde{U} \geq \bar{U}$, since the latter is found by optimizing over all possible $\beta : \hat{X} \to [0, 1]$, including the restriction of $\beta_*(\cdot)$. Thus,

$$\bar{U}(\boldsymbol{s}) - U(\boldsymbol{s}) \;\leq\; \tilde{U}(\boldsymbol{s}) - U(\boldsymbol{s}) \;=\; \sum_{\boldsymbol{x}\in\Xi} \tilde{\alpha}(\boldsymbol{x})\,(q(\boldsymbol{x}) - U(\boldsymbol{x})) \;\leq\; \sum_{\boldsymbol{x}\in\Xi} \alpha(\boldsymbol{x})\,(q(\boldsymbol{x}) - U(\boldsymbol{x}))\,,$$

which completes the proof of part 1. To prove part 2, select $q(\boldsymbol{x}) = C$, $\forall\, \boldsymbol{x} \in \Xi$. Since the exit-penalty C is prohibitively high, the stochastic path starting from $\boldsymbol{s} \in \hat{X}$ and using the optimal routing policy will avoid Ξ with probability 1. Thus, $\bar{U}(\boldsymbol{s}) = \hat{U}(\boldsymbol{s})$ (see the last part of Observation 1), and using the above result

$$\hat{U}(\boldsymbol{s}) - U(\boldsymbol{s}) \;\leq\; \sum_{\boldsymbol{x}\in\Xi} \alpha(\boldsymbol{x})\,(C - U(\boldsymbol{x})) \;\leq\; C \sum_{\boldsymbol{x}\in\Xi} \alpha(\boldsymbol{x}).$$

□

Let $d(\boldsymbol{x})$ be the distance from $\boldsymbol{x}$ to the characteristic passing through $\boldsymbol{s}$ (i.e., the $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory).

Conjecture 1. *There exists a constant $\rho > 0$ such that, for small enough h, $\alpha(\boldsymbol{x}) \leq e^{-\rho[d(\boldsymbol{x})]^2/h}$.*

As of right now, we only have a rigorous proof of this statement for an upwind discretization of a constant-coefficient advection PDE [16, Chapter 4]. The same proof also covers the Eikonal equation when all characteristics are parallel, but this clearly does not hold for the case $Q = \{\boldsymbol{t}\}$. Still, the numerical evidence (see Figure 2.18) indicates that this exponential decay is also present in the current context as well.

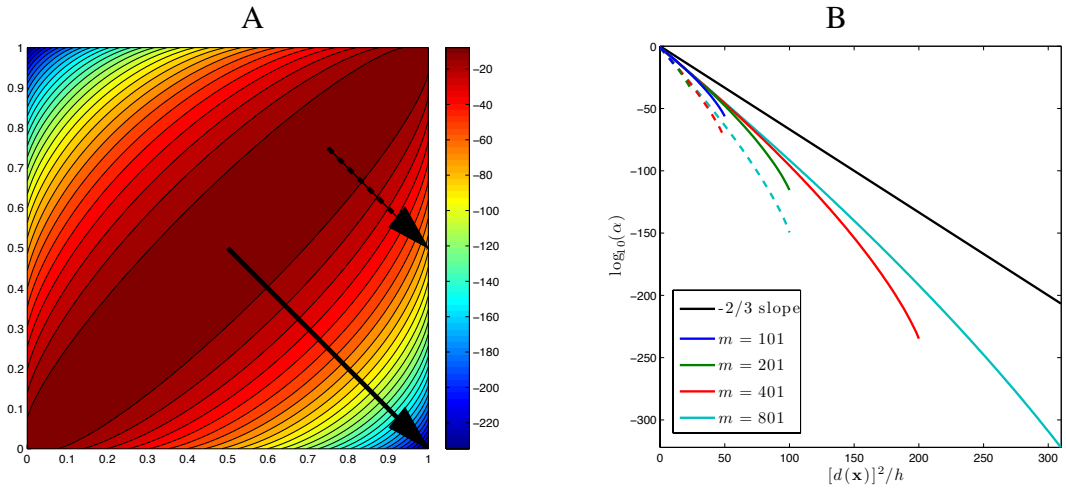


Figure 2.18: Alpha values decaying away from the characteristic. Subfigure **A:** shows the level sets of $\log_{10}(\alpha)$ for the constant speed example considered in §2.4.1. The solid and dashed arrows are perpendicular to the $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory. Subfigure **B** shows the rate of decay of $\log_{10}(\alpha)$ along each of these arrows computed for several different grid resolutions.

Theorem 1. *Let $\{X^h\}$ be a family of Cartesian grids on Ω with gridsize $h = 1/(m-1)$ such that both $\boldsymbol{s}$ and $\boldsymbol{t}$ are gridpoints for all m. Define $\hat{X}^h = \{\boldsymbol{x} \in X^h \mid d(\boldsymbol{x}) < r\}$, where $r = O(h^\mu)$, for some $\mu \in [0, \frac{1}{2})$. Let U^h and $\hat{U}^h$ be numerical solutions of the system (2.6) on X^h and $\hat{X}^h$ respectively. If Conjecture 1 holds, then $\left(\hat{U}^h(\boldsymbol{s}) - U^h(\boldsymbol{s})\right) \to 0$ as $h \to 0$.*

Proof. We note that the n-volume of the optimal-trajectory-centered r-cylinder approaches zero, though the total number of gridpoints in $\hat{X}^h$ grows as $h \to 0$. For convenience, we also define $k = r^2/h = O(h^{2\mu-1})$, which tends to $+\infty$ as $h \to 0$. If S is the path length of the $(\boldsymbol{s}, \boldsymbol{t})$-optimal trajectory, then the number of gridpoints in Ξ^h is $O(\frac{S r^{n-2}}{h^{n-1}}) = O(k^\nu)$, where $\nu = \frac{(n-1)-\mu(n-2)}{1-2\mu} > 0$. Considering the shortest grid-aligned and $\hat{X}^h$-constrained path from any $\boldsymbol{x} \in \hat{X}^h$ to $\boldsymbol{t}$, it is easy to show that $D^h = (S+r)\sqrt{n}$ is the upper bound for that path's length. Thus, $C^h = D^h/F_1$ is an upper bound for $\max_{\boldsymbol{x} \in \hat{X}^h} \hat{U}^h(\boldsymbol{x})$. If Conjecture 1 holds, then asymptotically $\alpha^h(\boldsymbol{x}) \le e^{-\rho r^2/h} = e^{-\rho k}$, for all $\boldsymbol{x} \in \Xi_h$. By Lemma 2, $\left(\hat{U}^h(\boldsymbol{s}) - U^h(\boldsymbol{s})\right)$ is bound from above by $\left[C^h \sum_{\boldsymbol{x} \in \Xi^h} \alpha^h(\boldsymbol{x})\right] = O(k^\nu e^{-\rho k})$, which

converges to 0 under grid refinement. □

2.6 Conclusions

We have described a new A*-type modification of the Fast Marching Method solving Eikonal equations for a 'single source/target' problem. Unlike the prior methods for this problem, which were developed to mirror in the 'standard A*' algorithm on graphs [32], our approach is based on a lesser known 'alternative A*' [8]. These prior SA*-FMM methods [26, 55, 57, 58, 59, 84, 85] either introduce additional errors that vanish slowly (if at all) under grid refinement, or must accept a much larger portion of the domain. In contrast, our AA*-FMM is able to significantly restrict computations, with any additional errors quickly decreasing under grid refinement.

One weakness of AA*-FMM is the reliance on an overestimate Ψ, especially when the feasibility of any $(\boldsymbol{s}, \boldsymbol{t})$ trajectory is in question (e.g., in the presence of obstacles). A good Ψ can be also found from related control problems or based on Pontryagin Maximum Principle (PMP). Here we mention two more approaches not tested in the current paper:

- One can use $\Psi = \zeta U^C(\boldsymbol{s})$, where $\zeta > 1$ and U^C is the solution found by FMM on a much coarser grid.
- One can also use the output of SA*-FMM on the same grid with an aggressive/inconsistent φ, setting $\Psi = U^*(\boldsymbol{s})$.

In the latter case, AA*-FMM should be viewed as a post-processing technique to improve the accuracy. This might seem superfluous: after all, PMP could also be applied using the output of SA*-FMM as an initial guess. But as we show in Figures 2.11 and

2.16, the errors from SA*-FMM are likely to result in PMP converging to some other (locally, rather than globally) optimal trajectory.

The effectiveness of the AA* domain restriction depends on the *quality* of φ and Ψ. If the initial Ψ is overly conservative, it can also be improved dynamically using the Branch & Bound techniques. No benchmarking results for the latter approach were included here for the sake of brevity.

We also list several desirable future extensions with significant impact on applications. First, AA* can be used instead of SA* within D* and E* path replanners [26, 60]. Second, the original AA* on graphs is applicable in both label-setting and label-correcting algorithms. It should not be hard to incorporate the same idea into other non-iterative and fast iterative methods for Hamilton-Jacobi PDEs. Our preliminary results for the Locking Sweeping Method [3] prove the feasibility of this approach. Third, since many gridpoints will never be used, allocating memory for the entire grid may be wasteful (particularly in high dimensions). One approach, described in [57, 58, 59], is to allocate gridpoints as needed and make use of a hash lookup table. Our current implementation of AA*-FMM does not use this idea, but we hope to explore it in the future. Finally, we note that all of the A* techniques can be also trivially extended to problems with a single-source and multiple targets. Similar underestimates can be also built for a moderately large set of sources $\{\boldsymbol{s}_i\}$ (e.g., $\varphi = \min_i \varphi_i^0$).

The error analysis in §2.5 relies on a conjecture, which so far has been only proven for a linear advection equation. For the Eikonal case, we currently rely on experimental/numerical confirmation. Nevertheless, we believe that a similar approach will be also useful in analyzing errors in more general domain restriction problems; e.g., for the errors due to an "almost causal" domain decomposition in [13].

CHAPTER 3

ANYTIME A* FOR EIKONAL EQUATIONS

3.1 Introduction

Optimal path planning is one of the most common problems in robotics and artificial intelligence literature. It can be posed both in discrete setting (e.g., a search for a shortest or quickest path on a directed graph) and in continuous setting (e.g., a search for a quickest trajectory within some inhomogeneous domain $\Omega \subset \mathbb{R}^n$). A naive graph-discretization of a continuous state space is typically sufficient if one only cares about the 'reachability" (does there exist any path from the source $\boldsymbol{s}$ to the target $\boldsymbol{t}$?) or if it is enough to find an optimal graph-constrained path. Thus, in many applications it is preferable to formulate the optimality equation before discretizing the state space and then use efficient numerical methods to approximate the solution on a relevant part of Ω. Several such methods were developed in the last 20 years, mirroring the ideas of classical graph-theoretic algorithms. In this paper, we continue this line of work by developing *"anytime" trajectory planning* methods for the continuous setting.

On graphs, the dynamic programming approach reduces path planning to finding the value function $U(\boldsymbol{x}_i)$ describing the minimal time to the target node $\boldsymbol{t}$ from each starting node $\boldsymbol{x}_i$. If the link transition penalties are non-negative, the value function can be computed by Dijkstra's classical algorithm [22], whose output can be used to recover the quickest path to $\boldsymbol{t}$ from every $\boldsymbol{x}_i$. However, if the optimal path is needed for one specific starting node $\boldsymbol{s}$ only, U values on most of the graph are likely irrelevant. A*-methods [32, 34, 35, 62, 80] achieve higher efficiency by focusing the U-computations on the

This chapter is in preparation for publication and will be submitted as *Anytime A* for Eikonal Equations* by Z. Clawson.

neighborhood of that ($\boldsymbol{s} \rightarrow \boldsymbol{t}$)-optimal path rather than on the full graph. However, for some applications even A* techniques might be insufficiently fast; e.g., when it is necessary to frequently replan the path online (as soon as new information becomes available). *Anytime A** algorithms [46, 7, 30, 31, 64] are particularly suitable in such cases since they yield a preliminary suboptimal path much earlier and iteratively improve it (until the truly optimal path is found) if more runtime is available.

In the continuous case, the value function $u : \bar{\Omega} \rightarrow \mathbb{R}$ is typically recovered as a viscosity solution of the corresponding Hamilton-Jacobi equation. Assuming that the dynamics is isotropic, that equation reduces to an Eikonal PDE

$$\begin{cases} |\nabla u(\boldsymbol{x})|\, f(\boldsymbol{x}) &= 1, \quad \boldsymbol{x} \in \bar{\Omega} \backslash \{\boldsymbol{t}\} \\ u(\boldsymbol{t}) &= 0. \end{cases}, \tag{3.1}$$

where $f(\boldsymbol{x})$ is the speed of motion through the point $\boldsymbol{x}$, and $u(\boldsymbol{x})$ is the minimal time from $\boldsymbol{x}$ to $\boldsymbol{t}$. This PDE is then discretized on a grid or a mesh, resulting in a coupled system of non-linear equations. With a suitable discretization, it is possible to recover the approximate solution non-iteratively in a label-setting manner. Two such Dijkstra-like methods were introduced for first order accurate discretizations of (3.1) on cartesian grids by Tsitsiklis [81] and Sethian [67]. Sethian's Fast Marching Method (FMM) was later extended to simplicial meshes in $\mathbb{R}^n$ and on manifolds as well as for higher order discretizations [41, 69, 68]. Related non-iterative methods were also developed for the more general anisotropic optimal control problems [70, 71, 2, 52, 53].

All of the above techniques approximate u on the entire Ω and are not very efficient if we are only interested in one specific starting position $\boldsymbol{s} \in \Omega$. In the last 10 years, several A*-FMM algorithms were developed for this problem, restricting the computational domain to a neighborhood of the ($\boldsymbol{s} \rightarrow \boldsymbol{t}$)-optimal trajectory [26, 55, 57, 58, 59, 84, 85, 18]. Many of these methods introduce a non-trivial trade-off: the aggressiveness of domain restriction improves the run-time, but, unlike on graphs, it may also introduce an addi-

tional approximation error. A careful analysis of the consequences for the convergence was completed only recently in [18].

In this paper, we leverage the A*-FMM methods developed in [18] and [84, 85] combining them with the anytime planning techniques developed on graphs in [46] and [7]. The resulting methods take more time than A*-FMM to find the truly optimal trajectories, but take less time to produce reasonable suboptimal trajectories needed in online planning applications. (Of course, the same trade off is also present in anytime path planning on graphs.)

We start by reviewing the relevant algorithms on graphs in section 3.2. We then describe the continuous isotropic optimal trajectory problem and the related numerical methods for solving (3.1) in section 3.3. We test the two versions of Anytime A*-FMM on examples described in section 3.4. The limitations and the related open questions are discussed in section 3.5.

3.2 Shortest Paths on Graphs

Consider a directed graph $\mathcal{G}$ on the nodes $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M, \boldsymbol{t} = \boldsymbol{x}_{M+1}\}$ with positive link weights (or transition time penalties) $C_{ij} = C\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) > 0$. We will use the convention that $C_{ij} = +\infty$ whenever there is no arc from $\boldsymbol{x}_i$ to $\boldsymbol{x}_j$. For convenience we will also define for each node the sets of its *in-neighbors* and *out-neighbors*:

$$N_i^- = N^-(\boldsymbol{x}_i) = \left\{\boldsymbol{x}_j \mid C_{ji} < +\infty\right\}, \qquad \text{and} \qquad N_i^+ = N^+(\boldsymbol{x}_i) = \left\{\boldsymbol{x}_j \mid C_{ij} < +\infty\right\},$$

with an assumption that $\max_i |N_i^+| = \kappa \ll M$. The value function $U_i = U(\boldsymbol{x}_i)$ represents the minimum total time needed to travel node-to-node from $\boldsymbol{x}_i$ to $\boldsymbol{t}$. Clearly $U(\boldsymbol{t}) = 0$,

and by *Bellman's Optimality Principle* [6]

$$U_i \;=\; \min_{\boldsymbol{x}_j \in N_i^+} \left\{ U_j + C_{ij} \right\}; \qquad i = 1, \ldots, M. \tag{3.2}$$

Once all U values are known, an optimal $(\boldsymbol{x}_i, \boldsymbol{t})$ path can be recovered recursively, with the first step from $\boldsymbol{x}_i$ leading to the minimizing neighbor $\boldsymbol{x}_{j_*}$ found in (3.2).

The above coupled system enjoys two useful properties:

- *monotonicity* (U_i is a monotone non-decreasing function of U_j, $\forall \boldsymbol{x}_j \in N_i^+$) and
- *causality* (if $U_i < U_j < K$, then plugging in K instead of U_j in (3.2) would not change U_i).

Dijkstra's algorithm [22] exploits these properties and can be used to find $U(\boldsymbol{x})$ for all $\boldsymbol{x} \in X$ efficiently. The algorithm operates by maintaining two lists: **considered** (or "open") nodes, whose value has been already estimated but not yet finalized (CONSIDERED); and **accepted** (or "closed" nodes) that have received a final and permanent value (ACCEPTED). In the main loop of the algorithm, the node $\bar{\boldsymbol{x}}$ is selected as the minimizer of U among all nodes currently on CONSIDERED; $\bar{\boldsymbol{x}}$ is then moved to ACCEPTED; all of its not-yet-accepted in-neighbors are updated and placed on CONSIDERED. Each node's value is updated at most κ times before it is accepted and all nodes placed on ACCEPTED never re-enter CONSIDERED. Algorithms with this property are called "single-pass" or "label-setting". To facilitate the selection of $\bar{\boldsymbol{x}}$, the list CONSIDERED is often implemented as a binary heap, resulting in a $O(M \log M)$ computational complexity.

Dijkstra's and the related methods discussed below are summarized in Algorithm 3.1. To keep the pseudocode general, it is written using special placeholders (SPECIALINIT, NOTYETDONE, SORTKEY, UPDATERELEVANT, and ADDRELEVANT) with their meaning specified for each method in Table 3.1.

Algorithm 2: *Dijkstra's Algorithm*

Initialization:

$U_i \leftarrow +\infty$ and mark $\boldsymbol{x}_i$ as FAR for $i = 1, 2, \ldots, M$
$U(\boldsymbol{t}) \leftarrow 0$ and CONSIDERED $\leftarrow \{\boldsymbol{t}\}$.
SPECIALINIT

Algorithm :

```
while NOTYETDONE do
    x̄ ← node in CONSIDERED with smallest SORTKEY
    Move x̄ from CONSIDERED to ACCEPTED
    for x_i ∈ N_j^-(x̄) such that U_i > U(x̄ + C(x_i, x̄)) do
        if x_i is UPDATERELEVANT then
            U_i ← U(x̄) + C(x_i, x̄)
            if x_i is ADDRELEVANT then
                Move x_i into CONSIDERED
```

Figure 3.1: Pseudocode for algorithms described in Table 3.1.

	Dijkstra's [22]	**AA* [8]**	**SA* [32, 35]**	**SA* [34, 80]**	**WA* [62]**	**AWA* [30, 31]**
Parameters	–	$\Psi \geq U(\boldsymbol{s})$	–		$w \geq 1$	
SPECIALINIT	–	$U(\boldsymbol{s}) \leftarrow \Psi$	–			
NOTYETDONE	CONSIDERED $\neq \emptyset$	$\boldsymbol{s} \notin$ ACCEPTED				CONSIDERED $\neq \emptyset$
SORTKEY	U	U	$U + \varphi$		$U + w\varphi$	
UPDATERELEVANT	$\boldsymbol{x}_i \notin$ ACCEPTED			all nodes	$\boldsymbol{x}_i \notin$ ACCEPTED	all nodes
ADDRELEVANT	all nodes	$U_i + \varphi_i \leq U(\boldsymbol{s})$	all nodes			$U_i + \varphi_i \leq U(\boldsymbol{s})$
φ has to be	–	admissible	consistent	admissible	consistent	
Method type	LS			LC	LS	LC
Output correct on	X	ACCEPTED		$(\boldsymbol{s}, \boldsymbol{t})$ opti path	–	X

Table 3.1: Dijkstra's and related A*-type algorithms.

3.2.1 A* Modifications

The output from Dijkstra's algorithm produces the correct U-value at every node in the graph, with many of them irrelevant if we only need an optimal path to $\boldsymbol{t}$ from a specific node $\boldsymbol{s}$. Once $\boldsymbol{s}$ is accepted, Dijkstra's algorithm can be terminated[1], but this may not

[1] In fact this termination condition was also suggested in Dijkstra's original paper [22].

provide much computational savings. Heuristic-reliant A* algorithms were introduced to improve the efficiency in this case. If we define the *reverse value function*

$$V(\boldsymbol{x}_i) \quad = \quad \text{minimum time from } \boldsymbol{s} \text{ to } \boldsymbol{x}_i,$$

then $(U_i + V_i) = U(\boldsymbol{s}) = V(\boldsymbol{t})$ provided $\boldsymbol{x}_i$ lies on an optimal path from $\boldsymbol{s}$ to $\boldsymbol{t}$ and $(U_i + V_i) > U(\boldsymbol{s}) = V(\boldsymbol{t})$ otherwise. This suggests that only those parts of the graph where $(U + V)$ is low are actually relevant. Since this reverse value function V is a priori unavailable, A* algorithms rely on a heuristic function $\varphi : X \to [0, +\infty)$ instead. This heuristic is *admissible* if it is an underestimate, i.e., $\varphi \leq V$ on X. An admissible heuristic φ is called *consistent* provided $\varphi_j \leq \varphi_i + C_{ij}$ for all $\boldsymbol{x}_i, \boldsymbol{x}_j \in X$. In many applications, such heuristics are readily available (e.g., based on the airplane distance if the graph represents a road network).

In [32] it was shown that, for a consistent heuristic, sorting CONSIDERED based on $(U + \varphi)$ instead of (U) still ensures that all accepted nodes have correct values. However, if φ is merely admissible, the accepted values may not be correct; thus, it becomes necessary to update and re-insert them to CONSIDERED later on. (This type of method is often called 'label-correcting".) The method still terminates as soon as $\boldsymbol{s}$ is accepted the first time, but the U values are only guaranteed to be correct along an $(\boldsymbol{s} \to \boldsymbol{t})$-optimal path. Table 3.1 reflects this change in character of the method depending on the properties of φ.

We refer to the above A*-algorithms as "standard" (SA*). An "alternative" version (AA*) was introduced in [8] and relies on an additional parameter Ψ that overestimates the value of $U(\boldsymbol{s})$. (Ψ is often obtained using some known $(\boldsymbol{s}, \boldsymbol{t})$-suboptimal path.) As in Dijkstra's, the CONSIDERED nodes are sorted based on U, but Ψ is used as the initial value for $U(\boldsymbol{s})$ and the nodes are only added to CONSIDERED if their $(U + \varphi)$ does not exceed the current value of $U(\boldsymbol{s})$. Unlike SA*, this is simply a pruning technique; since in AA* φ

is not used for sorting, φ only needs to be admissible and all accepted values are correct. As described in [8], AA* is a label-setting method, though the same pruning technique can be also used in all label-correcting algorithms (e.g., Bellman-Ford, D'Esopo-Pape, etc).

The computational cost of SA* depends on the accuracy of the heuristic φ. E.g., when $\varphi \equiv 0$ it is equivalent to Dijkstra's, but with $\varphi \equiv V$ it accepts only nodes on optimal $(\boldsymbol{s}, \boldsymbol{t})$-paths. Realistic field applications for shortest path problems provide a small timeframe for these algorithms to produce a solution. For such problems, the computational cost of SA* might be still prohibitive, and a good suboptimal path could be more useful if it is found quickly. This motivates a faster **Weighted A*** label-setting algorithm (WA*) [62], where a known consistent heuristic φ is "inflated" by a constant factor $w > 1$. The resulting heuristic $\varphi^w(\boldsymbol{x}) = w\varphi(\boldsymbol{x})$ may not be admissible; as a result, the output of WA* does not solve (3.2), but is bound from above by the w-scaled true solution.

3.2.2 Anytime A* Algorithms

WA* can be used to produce an $(\boldsymbol{s} \to \boldsymbol{t})$-path very quickly and also yields the bound on the suboptimality of the found path. It is natural to ask how that path can be further improved if more computational time is available. This can be accomplished by either (1) continuing to run the algorithm even after $\boldsymbol{s}$ is removed from CONSIDERED or (2) repeatedly re-running the algorithm with a different (more accurate) sorting criterion for CONSIDERED. These options give rise to the different *Anytime A* algorithms* described below:

- [1997, 2007] **Anytime Heuristic Search** (AHS) is due to Hansen et al. [30,

31]. AHS is a general approach for creating 'anytime' version of any admissible heuristic search algorithm (including A*).

For example, they suggest an *Anytime Weighted A** (AWA*) algorithm that essentially operates as a label-correcting version of WA* (for a fixed w) with pruning similar to AA*. A new/improved path is found whenever $U(\boldsymbol{s})$ is updated, and the algorithm terminates when CONSIDERED becomes empty. This algorithm is also represented by Pseudocode 3.1 and summarized in Table 3.1.

- [2003] **Anytime Repairing A*** (ARA*) is due to Likhachev et al. [46]. In ARA*, WA* is called *iteratively*, initially selecting $w = w_0$ and then decreasing w at the end of each iteration. (We refer to this approach as an "iterative label-setting".) The *NotYetDone* condition used in each iteration is modified to ensure that

$$\min_{\boldsymbol{x}\in\text{CONSIDERED}} \{U(\boldsymbol{x}) + w \cdot \varphi(\boldsymbol{x})\} \leq \Psi. \tag{3.3}$$

This results in an earlier termination when w is over-inflated. No pruning is performed in their algorithm (i.e., all nodes not-yet-accepted in the current iteration are *AddRelevant*). Additionally, instead of simply decreasing w by a constant Δw, they suggest taking

$$w \leftarrow \min\left\{w - \Delta w,\ U(\boldsymbol{s}) \,/\, \min_{\boldsymbol{x}\in\text{CONSIDERED}} \{U(\boldsymbol{x}) + \varphi(\boldsymbol{x})\}\right\} \tag{3.4}$$

- [2011] **Anytime Nonparametric A*** (ANA*) is due to van den Berg et al. [7]. This algorithm was designed to avoid the need for selecting good values of parameters $(w_0, \Delta w)$, which are used in ARA*. The unique aspect of ANA* is its $SortKey$. Based on (3.3), the authors suggest selecting the CONSIDERED node with *maximal* w, i.e.

$$w(\boldsymbol{x}) \leq \frac{\Psi - U(\boldsymbol{x})}{\varphi(\boldsymbol{x})}. \tag{3.5}$$

Each iteration ends when $\boldsymbol{s}$ becomes accepted (and Ψ is updated) or when CONSIDERED becomes empty. In the pseudocode provided in [7], the authors appear

to be using label-correcting on each iteration (i.e., all nodes are *UpdateRelevant*). The authors also use the pruning technique suggested by [30, 31] both *during* the iterations and *in between* iterations (as a post-processing technique, whenever Ψ changes).

Since both ARA* and ANA* use an outer loop for calling a Dijkstra-like method, these algorithms are not covered by the Pseudocode 3.1. Instead, we summarize their properties in Table 3.2 and provide pseudocodes for their "continuous domain versions" in §3.3 Figures 3.3, 3.4, 3.2.

	AWA* [30, 31]	**ARA* [46]**	**ANA* [7]**
Parameters	$w \geq 1$ fixed	$w \geq 1$, $\Delta w \geq 0$	$\Psi \geq U(\boldsymbol{s})$
SortKey	$U + w\varphi$	$U + w\varphi$	$-(\Psi - U)/\varphi$
Iter. NotYetDone	–	$\boldsymbol{s} \notin$ ACCEPTED AND CONSIDERED $\neq \emptyset$ AND formula (3.3)	$\boldsymbol{s} \notin$ ACCEPTED AND CONSIDERED $\neq \emptyset$
After each iter.	–	formula (3.4)	$\Psi \leftarrow U(\boldsymbol{s})$
Alg. termination	CONSIDERED $= \emptyset$	$w\varphi$ consistent OR CONSIDERED $= \emptyset$	CONSIDERED $= \emptyset$
Uses pruning	yes	no	yes
Method type	LC	iterative LS	iterative LC

Table 3.2: Original settings for Anytime A*-type algorithms on graphs. The pseudocode for their continuous counterparts is provided in Figures 3.3, 3.4, 3.2.. (AWA* is included both here and in Table 3.1 for the sake of completeness.)

3.3 Continuous Optimal Trajectories

Given an open set $\Omega \subset \mathbb{R}^n$ and a target point $\boldsymbol{t} \in \bar{\Omega}$, it is often necessary to find a quickest path from every starting position $\boldsymbol{x} \in \bar{\Omega}$ to the target. If the controlled dynamics are isotropic (i.e., the speed of motion $f > 0$ depends only on the current position), it is easy to show that

- the min-time-to-$\boldsymbol{t}$ value function $u : \bar{\Omega} \to [0, +\infty]$ is Lipschitz-continuous on the set where $u(\boldsymbol{x})$ is finite (i.e., for all $\boldsymbol{x}$ in the same connected component of $\bar{\Omega}$ as $\boldsymbol{t}$);
- wherever u is smooth, it satisfies the Eikonal equation (3.1);
- for every point $\boldsymbol{x} \in \bar{\Omega}$ where ∇u is well-defined, the optimal direction of motion toward $\boldsymbol{t}$ is $\boldsymbol{a}^*(\boldsymbol{x}) = -\nabla u(\boldsymbol{x}) / |\nabla u(\boldsymbol{x})|$.

Thus, using a numerical method to approximately solve (3.1) is all that is needed to find time optimal trajectories from all starting positions. Several technical challenges need to be addressed to make this approach practical:

- The value function u is typically not smooth on the entire $\bar{\Omega}$. The points where u lacks differentiability are precisely the starting positions from which an optimal trajectory is not unique.
- Similarly, it is well-known that (3.1) and the related PDEs usually do not have "classical" solutions. Moreover, the weak solutions (i.e., Lipschitz-continuous functions satisfying (3.1) almost everywhere) are not unique. To circumvent this, additional pointwise test conditions were introduced by Crandall and Lions [19] to focus on the *viscosity solution* – the unique weak solution coinciding with the value function of the associated optimal control problem. The numerical methods discussed below are based on discretizations that are consistent and monotone, two important properties used to prove that U converges to the viscosity solution u [5].
- The time optimal trajectories to $\boldsymbol{t}$ are constrained to remain in $\bar{\Omega}$. No boundary conditions are specified for the PDE on $\partial\Omega \setminus \{\boldsymbol{t}\}$ (i.e., a trajectory can continue along $\partial\Omega$). This leads to a similar notion of a "$\bar{\Omega}$-restricted viscosity solution" [4]. For the purposes of discretization, this is equivalent to assuming that $u = +\infty$ outside of $\bar{\Omega}$.

- In discretizing this PDE with finite differences, the value function at each point $\boldsymbol{x}$ will generally depend on the u values at n neighboring points straddling the optimal trajectory from $\boldsymbol{x}$ to $\boldsymbol{t}$. This is in contrast with the problems on graph, where $U(\boldsymbol{x})$ depended only on the value at a single neighbor (the next node on the $(\boldsymbol{x} \rightarrow \boldsymbol{t})$-optimal path). As we describe below this increased dependency set results in additional conditions for applicability of Dijkstra-like, A*, and Anytime-A* techniques in the continuous case.

For simplicity we will assume that $\bar{\Omega} = [0,1]^n \subset \mathbb{R}^n$ is discretized on a m^n cartesian grid X with grid-spacing $h = 1/(m-1)$. We will also assume that the speed function $f : \bar{\Omega} \rightarrow [0, +\infty)$ satisfies

$$0 \;<\; F_1 \;\leq\; f(\boldsymbol{x}) \;\leq\; F_2 \;<\; +\infty.$$

To simplify the notation, we will describe everything in 2D although the generalization to higher dimensions is straightforward. For a gridpoint $\boldsymbol{x}_{ij} = (ih, jh)$, the value function $u(\boldsymbol{x}_{ij})$ is approximated by $U_{ij} = U(\boldsymbol{x}_{ij})$, the speed is f_{ij}, etc. We also use $N(\boldsymbol{x}_{ij})$ to represent the "immediate neighbors" of the gridpoint $\boldsymbol{x}_{ij}$. (On a cartesian grid with the regular 4-point stencil, $N(\boldsymbol{x}_{ij})$ contains 4 adjacent gridpoints unless $\boldsymbol{x}_{ij} \in \partial\Omega$.) We will further assume that $\boldsymbol{s}$ and $\boldsymbol{t}$ are located at gridpoints. Using upwind finite differences, we can discretize (3.1) at every $\boldsymbol{x}_{ij} \in \Omega \setminus \{\boldsymbol{t}\}$ by

$$\left(\max\left\{D^{-x}U_{ij}, -D^{+x}U_{ij}, 0\right\}\right)^2 \;+\; \left(\max\left\{D^{-y}U_{ij}, -D^{+y}U_{ij}, 0\right\}\right)^2 \;=\; 1 \,/\, f_{ij}^2, \qquad (3.6)$$

$$\text{where} \quad u_x(x_i, y_j) \approx D^{\pm x}U_{ij} = \frac{U_{i\pm1,j} - U_{ij}}{\pm h}, \quad \text{and} \quad u_y(x_i, y_j) \approx D^{\pm y}U_{ij} = \frac{U_{i,j\pm1} - U_{ij}}{\pm h}.$$

(The same formula works when $\boldsymbol{x}_{ij} \in \partial\Omega$, provided we use the convention that $U(\boldsymbol{x}) = +\infty$ for all $\boldsymbol{x} \notin \bar{\Omega}$.) If all the neighboring values are already known, the formula (3.6) is really a "quadratic equation in disguise" for U_{ij}.

There are 8 expressions that (3.6) can evaluate to on $X \backslash \{\boldsymbol{t}\}$, each case defined by what the maxima produce on the left hand side. (The case where both maxima evaluate to zero is excluded since it would correspond to $f_{ij} = +\infty$.) In 4 of these 8 cases, both maxima evaluate to a positive quantity, where U_{ij} is produced using two neighboring gridpoints from a single quadrant. For example, if $-D^{+x}U_{ij}$ and $-D^{+y}U_{ij}$ attain the maxima then U_{ij} is produced using $U_E = U_{i+1,j}$ and $U_N = U_{i,j+1}$. The quadratic equation for this *two-sided* calculation reduces to

$$\left(U_E^2 + U_N^2\right) U_{ij}^2 \;-\; 2U_{ij}\left(U_E + U_N\right) \;+\; \left(U_E^2 + U_N^2 - h^2/f_{ij}^2\right) \;=\; 0. \tag{3.7}$$

However, to make this consistent with the choice of the north-eastern quadrant, both $-D^{+x}U_{ij}$ and $-D^{+y}U_{ij}$ should be at least non-negative; so, we select U_{NE} to be the smallest real solution of (3.7) satisfying the *upwinding condition*

$$U_{NE} \;\geq\; \max\left\{U_E, U_N\right\}. \tag{3.8}$$

If no such real root exists, we resort to a "one-sided-update" (corresponding to one of the 4 remaining cases), selecting the smaller of the 2 neighbors:

$$U_{NE} \;\leftarrow\; \min\left\{U_E, U_N\right\} \;+\; h/f_{ij}. \tag{3.9}$$

The updates from the other 3 quadrants are defined similarly, and U_{ij} is found by minimizing over all 4 quadrants:

$$U_{ij} \;\leftarrow\; \min\left\{U_{NE},\; U_{NW},\; U_{SW},\; U_{SE}\right\}. \tag{3.10}$$

This update formula can be further reduced to a simpler form on cartesian grids.

The above procedure can be used to find U_{ij} when all the neighboring grid values are already known. But since all values on $X \setminus \{\boldsymbol{t}\}$ are a priori unknown, this leads to a coupled system of m^2 equations[2]. Fortunately, the fact that U_{ij} depends only on neighbors with *smaller* U-values implies that a Dijkstra-like approach is applicable [81, 67].

[2]This coupled system can also be solved iteratively either using either a "fast sweeping" (i.e., Gauss-Seidel relaxation with alternating gridpoint orderings) method [12, 88] or a variety of label-correcting

The **Fast Marching Method** (FMM) operates by simply replacing Line 9 of Algorithm 1 with the above update procedure. On graphs, the update was always computed from the newly accepted neighbor $\bar{\boldsymbol{x}}$ only. Similarly, in FMM the update only needs to be computed *from the quadrant(s)* adjacent to $\bar{\boldsymbol{x}}$.

There are different versions of FMM in the literature based on whether the not-yet-accepted neighbors are used in producing an update at $\boldsymbol{x}_{ij}$ from a given quadrant. The choices are either to

- use their current/temporary values for all considered points [67];
- or assign $U(\boldsymbol{y}) = +\infty$ for all $\boldsymbol{y} \in N(\boldsymbol{x}) \cap$ CONSIDERED, essentially defaulting to a 1-sided update.

The efficiency of both approaches is largely the same when finding U on the entire X, but our implementation is based on the former since we found it to be much more efficient in the Anytime A* context.

Sethian's FMM was extended in many ways, including higher order discretizations and methods on simplicial meshes both on manifolds and in $\mathbb{R}^n$ [41, 69]. On unstructured meshes, the update is similarly computed to (3.10) on a simplex-by-simplex basis for all simplices that use $\boldsymbol{x}$ as a vertex. An *upwinding condition* generalizing (3.9) is then used to decide for each simplex if the full 'n-sided update' should be used or if the lower dimensional simplex should be considered instead [69, 84, 85]. In [69], it is shown that a Dijkstra-like method solves the resulting system of discretized equations correctly, provided the mesh does not include any obtuse simplices.

methods, including the two-scale variants that combine the label-setting/correcting ideas with sweeping; see [14, 15] and references therein. All these methods have their advantages on different sets of problems and their comparison with Dijkstra-like techniques for solving an Eikonal has been an active research area [29, 33, 14, 15]. In this paper we restrict the discussion to Dijkstra-like techniques because our real focus is on Anytime A* path-planning, which was developed in this context.

3.3.1 SA*-FMM and AA*-FMM

Both SA* [26, 55, 57, 58, 59, 84, 85] and AA* [18] can be used with FMM, but the analysis of the efficiency and accuracy is more subtle due to the fact that a gridpoint value U_{ij} generally depends on more than one neighbor.

These issues were analyzed in detail in [18], here we just provide the summary of the findings:

- For SA*-FMM, if the goal is to produce exactly the same value $U(\boldsymbol{s})$, this imposes a more complicated *consistency condition* on the underestimate φ.
- When SA*-FMM is used with an inconsistent heuristic this typically results in a wrong order of acceptance and additional error in $U(\boldsymbol{s})$; moreover, that additional error can be significantly larger than the discretization error and often decreases very slowly under grid refinement.
- Unfortunately, the only underestimate satisfying the consistency condition on the Cartesian grid is $\varphi \equiv 0$, resulting in the same set of accepted gridpoints as FMM and no computational savings.
- In [84, 85] it was shown that suitable consistent underestimates can be found on all strictly acute simplicial meshes.
- However, as explained in [18], such consistent underestimates on meshes tend to be very conservative and for many source/target pairs can result in the same set of accepted meshpoints as FMM.
- An alternative approach underlying AA* is to use an inconsistent heuristic but for pruning purposes only. This also introduces small (cf. SA*) additional numerical errors, but they decrease very quickly under grid refinement. The downside of

AA*-FMM is the need for an a priori overestimate Ψ and somewhat lesser computational savings than in SA*-FMM due to the less aggressive domain restriction.

In the context of Anytime planning, the absolute accuracy in computing $U(\boldsymbol{s})$ is far less important than producing a series of good approximations for it quickly. (Indeed, inflating a mesh-consistent underestimate φ by a factor $w > 1$ will usually make it inconsistent.) Therefore, in this paper we adopt the SA*-approach both on grids and meshes.

In all examples considered in §4, we use the *naïve heuristic* based on the *airline distance*

$$\varphi_0(\boldsymbol{x}) \;=\; |\boldsymbol{x} - \boldsymbol{s}| \;/\; F_2 \;\leq\; v(\boldsymbol{x}), \tag{3.11}$$

where $v(\boldsymbol{x})$ = minimum time from $\boldsymbol{x}$ to $\boldsymbol{s}$ is the unknown *reverse value function*. This heuristic was chosen because it is simple and cheap to compute in all problems.

3.3.2 Anytime A* Extensions

With all of the continuous framework in place, the Anytime A* algorithms can be easily extended to the continuous case with a few additional caveats. The pseudocode for the algorithms is provided in Figures 3.3, 3.4, 3.2, where ARA* and ANA* both make use of the "shared functions" on Lines 3–23. This implementation is close to the description provided in §3.2.2 and summarized in Table 3.2, but is different in several details described below.

Algorithm 3: *Shared functions for ARA* and ANA* algorithms*

```
procedure : INITIALIZE()
1  INCON ← ∅
2  ACCEPTED ← {t}
3  CONSIDERED ← N(t)
4  U(x) = +∞ ∀ x ∈ X
5  U(t) ← 0
6  U(y) ← h/f(y) ∀ y ∈ N(t)
7  Ψ ← +∞

procedure : PRUNE(CONSIDERED)
8  for x ∈ CONSIDERED do
9      if U(x) + φ(x) > Ψ then
10         Remove x from CONSIDERED

procedure : IMPROVESOLUTION()
11 while CONSIDERED ≠ ∅ do
12     x* ← BESTKEY()
13     Remove x* from CONSIDERED and put in ACCEPTED
14     if x* = x then
15         break
16     for y ∈ N(x*) do
17         if U(y) ≥ U(x*) then
18             U_temp ← UPDATE(y)
19             if U_temp < U(y) then
20                 U(y) ← U_temp if y ∈ ACCEPTED then
21                     Add y to INCON
22                 else if U(y) + φ(y) ≤ Ψ then
23                     Add y to CONSIDERED
```

Figure 3.2: Pseudocode for the shared functions that ARA* (Algorithm 3.3) and ANA* (Algorithm 3.4) use.

Algorithm 4: *ARA* Algorithm*

procedure : MAIN()
1 INITIALIZE() **while** $w\varphi$ *is inconsistent* **do**
2 IMPROVESOLUTION()
3 $\Psi \leftarrow U(\boldsymbol{s})$
4 Move INCON into CONSIDERED; ACCEPTED $\leftarrow \emptyset$
5 PRUNE(CONSIDERED)
6 $w \leftarrow \min\{w - \Delta w, U(\boldsymbol{s})/\varphi(\boldsymbol{t})\}$

procedure : SORTKEY($\boldsymbol{x}$)
7 **return** $U(\boldsymbol{x}) + w\varphi(\boldsymbol{x})$

procedure : BESTKEY()
8 **return** $\underset{\boldsymbol{x}\in\text{CONSIDERED}}{\text{argmin}}$ SORTKEY($\boldsymbol{x}$)

Figure 3.3: Pseudocode for ARA* algorithm.

Algorithm 5: *ANA* Algorithm*

procedure : MAIN()
1 INITIALIZE() **while** CONSIDERED $\neq \emptyset$ **do**
2 IMPROVESOLUTION()
3 $\Psi \leftarrow U(\boldsymbol{s})$
4 Move INCON into CONSIDERED; ACCEPTED $\leftarrow \emptyset$
5 PRUNE(CONSIDERED)

procedure : SORTKEY($\boldsymbol{x}$)
6 **return** $(\Psi - U(\boldsymbol{x}))/(\gamma h + \varphi(\boldsymbol{x}))$

procedure : BESTKEY()
7 **return** $\underset{\boldsymbol{x}\in\text{CONSIDERED}}{\text{argmax}}$ SORTKEY($\boldsymbol{x}$)

Figure 3.4: Pseudocode for ANA* algorithm.

Implementation details and differences from the discrete case:

- In essence, the main difference between the continuous and discrete setting is the UPDATE() procedure (3.10) (Line 18); to produce the correct output on graphs this should be replaced by Line 9 of Algorithm 3.1. The rest of the modifications on

this list were introduced for the sake of efficiency only.

- Here we present both algorithms as *iterative label-setting* methods, i.e. whenever a gridpoint is accepted it cannot become re-considered until the next iteration. This is different than the original ANA* [7], which was iterative label-correcting in nature [7]. To turn both algorithms into *iterative label-correcting* methods one could allow for previously accepted nodes to reenter the considered list (i.e., remove Lines 20–21 and change the following "**else if**" to "**if**" on Line 22).[3]

- *Pruning* (i.e. AA*) can be easily 'turned off' by substituting 0 for $\varphi(\boldsymbol{x})$ only on Lines 9 and 22.

- The ANA* SortKey from §2 (i.e. the righthand-side of (3.5)), would present and additional complication in the continuous setting. Here, the optimal path is a continuous curve through Ω rather than a node-to-node path. An optimal domain restriction algorithm would process gridpoints only in an ϵ-tube of this optimal trajectory, accepting them in ascending order based on U-values. Unfortunately the original ANA* SortKey does exactly the opposite in a neighborhood of $\boldsymbol{s}$. In fact, it prioritizes accepting $\boldsymbol{s}$ first since the SortKey is infinite there. Moreover, it has an even worse effect on gridpoints near $\boldsymbol{s}$ since the limit of the $(\Psi - U(\boldsymbol{x})/\varphi(\boldsymbol{x})$ is not well-defined as $\boldsymbol{x}$ approaches $\boldsymbol{s}$ (assuming $\Psi = U(\boldsymbol{s})$). To remedy this issue, we have used a modified SortKey on Line 6,

$$(\Psi - U) \,/\, (\gamma h + \varphi)\,, \tag{3.12}$$

  where $\gamma > 0$ is selected 'large enough' to correct this issue.[4]

[3] We have also implemented and tested this iterative label-correcting version but we found it far less efficient in this continuous setting. The iterative label-correcting methods suffer from a "back-propogation" of updates in the ImproveSolution() routine, where a single gridpoint triggers a chain reaction of gridpoint values becoming re-updated.

[4] Our experimental evidence shows that as γ decreases, so does the time between iterations. This is because decreasing γ increases the new key at $\boldsymbol{s}$, causing it to become ACCEPTED more quickly. γ can also be selected to control how quickly the iteration should terminate once $\boldsymbol{s} \in$ CONSIDERED.

- Unfortunately, if w_0 is too large and Δw is small, it might take many iterations until any meaningful improvement is discovered for the optimal $(\boldsymbol{s} - \boldsymbol{t})$-trajectory. On graphs the 'reset formula' (3.4) provided a way to decrease w to a 'smart' value. We use a slightly simpler version on Line 6 that that helps with efficiency given by

$$w \leftarrow \{w - \Delta w,\ U(\boldsymbol{s})/\varphi(\boldsymbol{t})\} \tag{3.13}$$

3.4 Numerical Results

In this section we conduct experiments on real and synthetic data both on cartesian grids and simplicial meshes. The performance of the Anytime A* algorithms is measured relative to the performance of SA*-FMM and classical FMM (terminated once $\boldsymbol{s} \in$ ACCEPTED). The Anytime methods are not quicker to produce the exact value $U(\boldsymbol{s})$, but this is not the goal of these algorithms. Instead, an Anytime A* algorithm is useful if it produces several high quality solutions in a fraction of the SA*-FMM total time T_{SA}, and with enough runtime produce the correct solution. However, in special situations SA*-FMM can take longer than FMM, so we compare to the total time $T_{total} = \min\{T_{SA}, T_{FMM}\}$. Typically $T_{SA} < T_{FMM}$, however when SA*-FMM provides little-to-no restriction it can be slower than regular FMM; e.g. see §3.4.5.

The main experimental evidence we present are performance profiles for ARA* and ANA*. If $U^*(\boldsymbol{s})$ is the currently found solution by a profiled method and $U(\boldsymbol{s})$ is the FMM-produced solution, we measure "solution quality" as the relative error at $\boldsymbol{s}$

$$\mathcal{E} \triangleq (U^*(\boldsymbol{s}) - U(\boldsymbol{s})) \ / \ U(\boldsymbol{s}) \geq 0.$$

Since SA*-FMM with an inconsistent φ can produce additional errors [18], we also report $\mathcal{E}^{SA}$ for all of the examples. In addition, we report $\mathcal{E}^{S}$, the measure of quality

for the straight-line trajectory from $\boldsymbol{s}$ to $\boldsymbol{t}$. An anytime method is worthwhile for a particular example if it produces several solutions with $\mathcal{E} \in \left[\mathcal{E}^{SA}, \mathcal{E}^{S}\right]$ before the time T_{total}. The results of benchmarking are succinctly summarized in Table 3.3. For anytime algorithms, $\mathcal{E}_\alpha$ refers to the error of the best solution found up to the time αT_{total}.

For each example we also provide *"time vs. solution quality"* plots in Figures 3.6, 3.9, 3.11, 3.13, and 3.16. $\mathcal{E}$ is capped at 1.0 and the curves are only plotted until the time $1.25T_{total}$.

Another important statistic is the size of SA*-FMM-accepted set compared to the FMM-accepted set, defined respectively by

$$L^* = \{\boldsymbol{x} \in X \mid U^*(\boldsymbol{x}) + \varphi(\boldsymbol{x}) \leq U^*(\boldsymbol{s})\} \qquad \text{and} \qquad L = \{\boldsymbol{x} \in X \mid U(\boldsymbol{x}) \leq U(\boldsymbol{s})\}.$$

Examples of particular interest are ones where SA*-FMM is unable to provide sufficient restriction, which is indicated when $|L^*| \,/\, |L|$ is closer to 1. This situation typically arises because $\varphi \ll V$ on much of Ω, and thus inflating φ helps provide more restriction; this justifies the use of WA*/ARA*/ANA*.

3.4.1 Experimental setup

The code package we developed was written in C++ and made use of the Eigen library. Due to the heavy use of template metaprogramming all results were compiled with `g++` in "Release mode", i.e. with compiler option `-O3`. All results were run on a 2014 Macbook Pro with a 2.8 GHz i7 processor and 16 GB RAM. We list the most important implementation details below:

- **ARA*** uses $(w_0, \Delta w) = (10, w_0/100)$ in all examples except for the last section. A

more efficient implementation could set w_0 heuristically as a function of h and/or f, which is done in Section 3.4.5 and further explained there.

- **ANA*** initializes $\Psi \leftarrow +\infty$. Although Ψ could be set to any finite value larger than $U(\boldsymbol{s})$, this decreases the aggressiveness of early iterations. In (3.12) we selected $\gamma = 1/10$, which worked well on all of our examples.
- **Pruning** is turned off for all results in this section. Our tests indicate that with it turned on:
  - The only iterations that benefitted from pruning were those occurring much later than T_{SA}, thus making it not useful.
  - Even though pruning benefits the later iterations, it also introduces additional AA*-type errors that go to zero under grid refinement [18].
- For consistency, f is precomputed and stored in memory for all examples.
- All results use an $m \times m$ cartesian grid with grid-spacing $h = 1/(m-1)$.

	References		**ARA* quality measures**				**ANA* quality measures**			
	$\mathcal{E}^{SA}$	$\mathcal{E}^{S}$	$\mathcal{E}_{1/8}$	$\mathcal{E}_{1/4}$	$\mathcal{E}_{1/2}$	$\mathcal{E}_{1}$	$\mathcal{E}_{1/8}$	$\mathcal{E}_{1/4}$	$\mathcal{E}_{1/2}$	$\mathcal{E}_{1}$
§3.4.2A	0.015	0.055	0.118	0.063	0.049	0.035	0.145	0.105	0.072	0.051
§3.4.2B	0.010	0.503	0.078	0.057	0.047	0.029	0.112	0.072	0.054	0.052
§3.4.3A	0.057	0.694	0.235	0.150	0.150	0.115	0.630	0.409	0.149	0.123
§3.4.3B	0.037	1.185	0.685	0.274	0.161	0.161	2.472	1.345	0.271	0.271
§3.4.4	0.001	1.385	0.049	0.048	0.039	0.036	0.386	0.067	0.043	0.037
§3.4.5	0.000	0.911	0.751	0.707	0.288	0.221	0.793	0.751	0.298	0.292
§3.4.6	0.193	N/A	0.426	0.426	0.426	0.319	0.443	0.443	0.345	0.281

Table 3.3: Statistics for f defined in §3.4.2–3.4.6, and the corresponding performance measures for ARA*. $\mathcal{E}^S$ is not available for the example in §3.4.6 due to the presence of impermeable obstacles. $\mathcal{E}_\alpha$ refers to the error of the best solution found up to the time αT_{total}.

3.4.2 Highly oscillatory sinusoid

Consider the two highly oscillatory sinusoidal speed functions

$$\begin{cases} f_A(x,y) &= 1 + 0.5\sin(10\pi x)\sin(10\pi y) \\ f_B(x,y) &= 2 + 1.99\sin(20\pi x)\sin(10\pi y)\,. \end{cases} \tag{3.14}$$

For both examples we use $\boldsymbol{s} = (0.9, 0.7) = (475h, 350h)$ and $\boldsymbol{t} = (0.3, 0.45) = (150h, 225h)$ for $m = 501$, and the solutions can be seen in Figure 3.5 (where color indicates the SA*-FMM solution). Visually, the quantity $|L^*|\,/\,|L|$ is much smaller in Figure 3.5A, which indicates a higher restriction rate. This is expected since φ_0 is much more accurate when used with f_A.

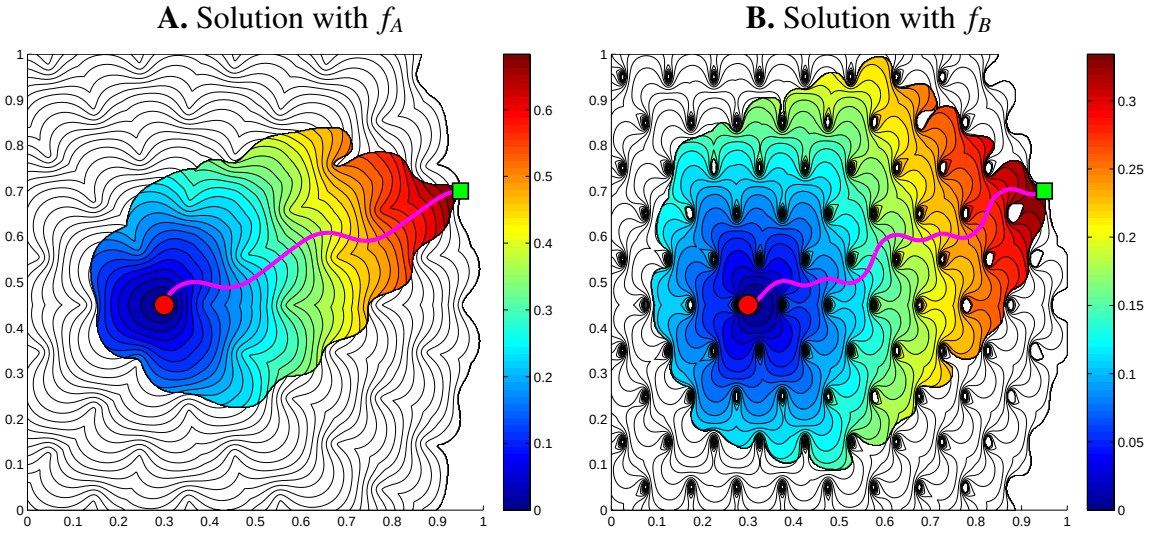


Figure 3.5: Contours of the solution to (3.1) using the speed functions f_A and f_B defined in (3.14). The sets L and L^* are shown by the contours, where color indicates the SA*-FMM restricted solution, and the curve from $\boldsymbol{s}$ to $\boldsymbol{t}$ represents the optimal path. he Anytime A* profiling plots for these two examples can be seen in Figure 3.6.

Figure 3.6 shows profiling plots for the anytime methods. An initial solution is found quickly and then iteratively improved, highlighting the usefulness of the algorithms. This can be confirmed by early availability of a large number of high-quality solutions

before the time T_{SA} occurs. In Figure 3.6A, ANA* errors appear to 'level off' at $\mathcal{E}_S$, but after $113T_{SA}$ seconds ANA* terminates with zero error. In comparison, ARA* takes $28T_{SA}$ seconds.

It is very clear that the performance of the anytime methods is significantly better on Example B; see Figure 3.6. Figure 3.5B suggests that the anytime algorithms have a better chance to make a difference in this example, which is confirmed by Figure 3.6B and the timing results in Table 3.3. Although, in both examples ARA* is able to produce high-quality solutions before T_{SA} at the benchmarked times in Table 3.3.

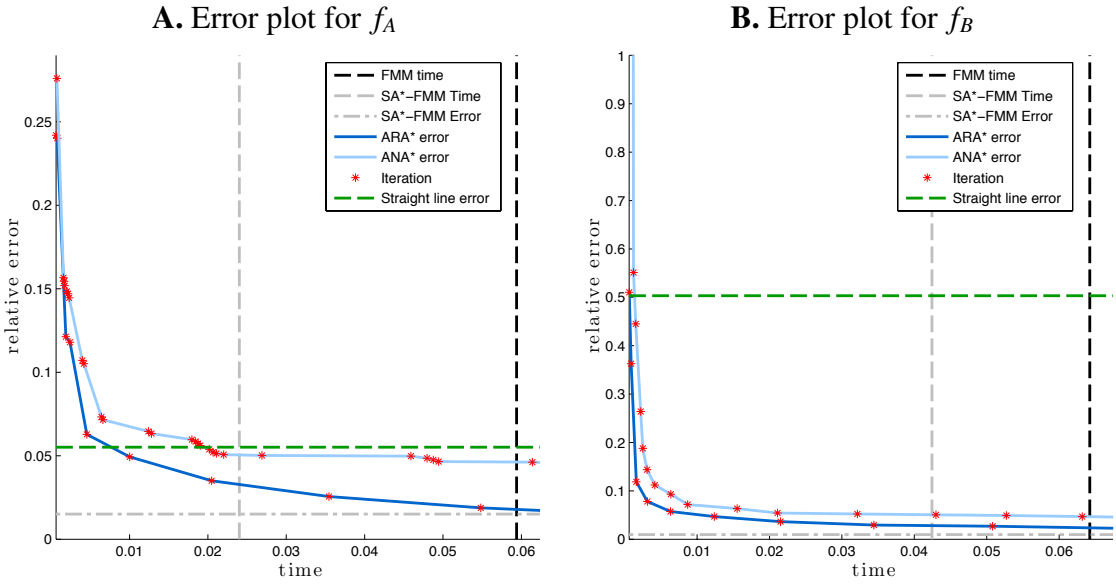


Figure 3.6: The relative error as the ARA* and ANA* algorithms progress when using the speeds defined in (3.14). See the level sets of the FMM solutions in Figure 3.5. Each * represents a single iteration of the Anytime A* algorithms.

3.4.3 Random checkerboard

We next explore the usefulness of anytime planning on problems with piecewise constant speed functions. The domain Ω is first discretized into a 328×328 grid, and then

decomposed into a $c \times c$ checkerboard of squares, where on each square the speed is randomly determined to be 'fast' or 'slow' as

$$F_{slow} = F_{1,N} = 1/2^N, \qquad F_{fast} = F_2 = 1. \tag{3.15}$$

In Figure 3.7C we show a checkerboard ($c = 41$) generated in this random way with the source and target shown by the square and circle markers respectively. Figures 3.7A&B show the corresponding solution for $N_A = 2$ and $N_B = 5$, respectively, with $m = 328$. All plots contain the optimal trajectory drawn from $\boldsymbol{s}$ to $\boldsymbol{t}$, and colored contours represent the solutions produced by SA*-FMM. SA*-FMM provides more restriction for Example A here due to the higher accuracy of φ_0. Note that given that $\boldsymbol{s}, \boldsymbol{t}$ are outside of the obstacles and F_2/F_1 is large enough, the optimal trajectory will never enter one of the obstacles.

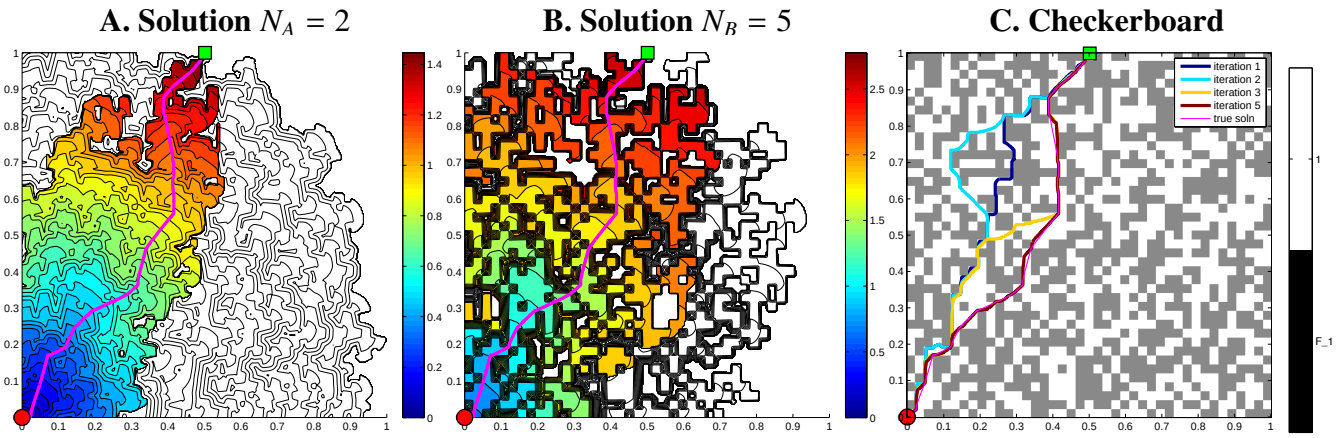


Figure 3.7: **A,B.** Contours of solutions to (3.1), where color indicates the SA*-FMM-accepted region. The FMM-recovered trajectory is drawn in all of the plots; the SA*-FMM-recovered trajectory is slightly perturbed from the FMM-trajectory, but the difference not visually noticeable and thus not shown. **C.** The randomly generated 41×41 checkerboard with the different ARA*-recovered trajectories plotted for $N = 5$.

Table 3.3 shows that both algorithms produce multiple suboptimal solutions before T_{SA}, although based on the plots and table it appears that the performance of the methods

on Example A is superior. The 'plateau' effect in the ANA* results (Figure 3.9B) is due to ANA* finding a new locally optimal solution and getting 'stuck' on it.

The different ARA*-produced suboptimal trajectories are shown in Figure 3.8 at the end of each iteration. ARA* completes in 26 iterations after $24T_{SA}$ seconds, in comparison to ANA*'s 281 iterations after $231T_{SA}$ seconds.

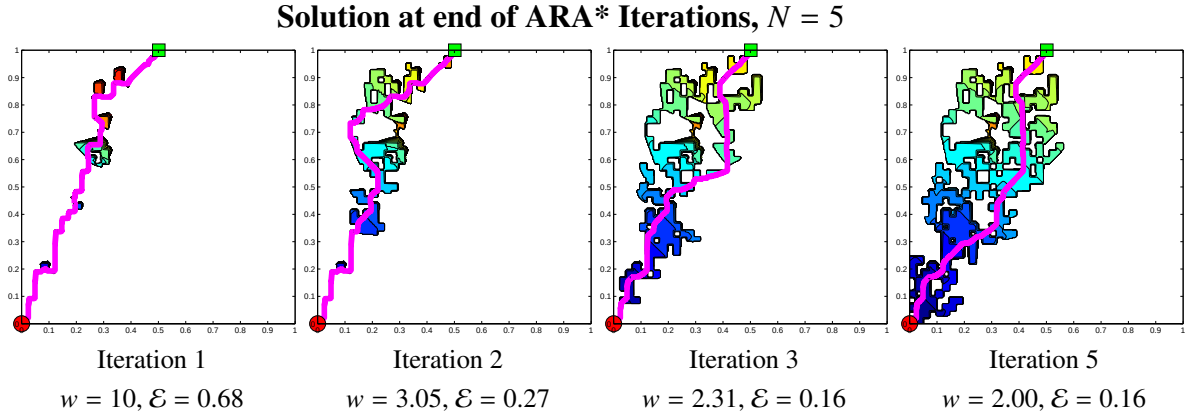


Figure 3.8: ARA*-produced suboptimal trajectories for specific iterations. The corresponding speed is explained in Figure 3.7C, along with corresponding solution in B. The trajectories can be compared to the error plot in Figure 3.9B: Iterations 1, 2, and 3 correspond to the first markers at each of the $\mathcal{E} \approx 0.68$, 0.27, and 0.16 levels in Figure 3.9B.

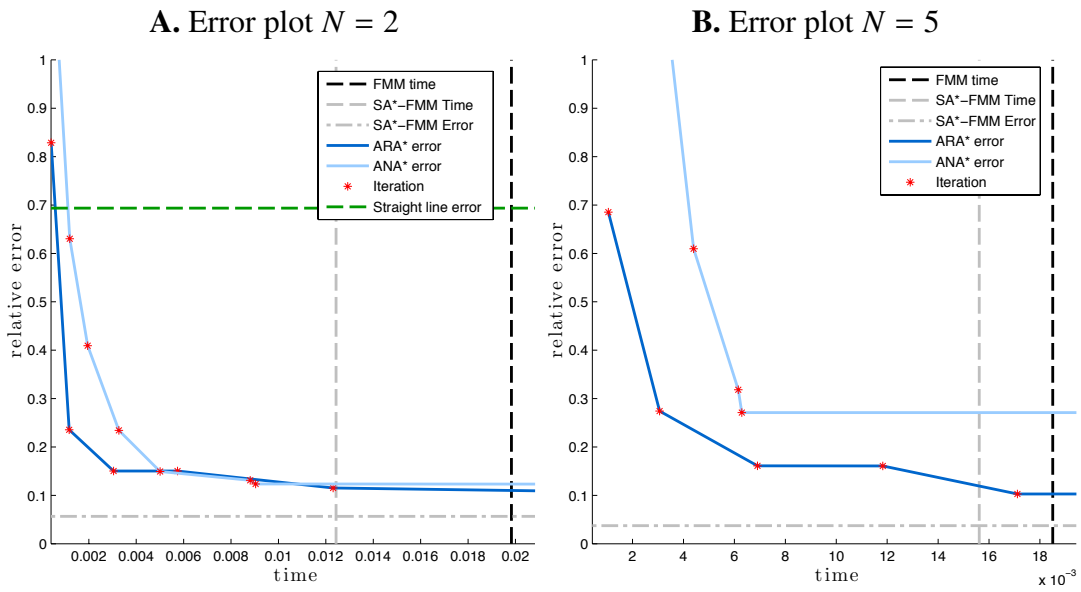


Figure 3.9: Anytime profiling plots corresponding to the 'random checkerboard' speed in Figure 3.7C for two different speeds on the slow checkers ($N = 2, 5$). The produced suboptimal trajectories are shown in Figure 3.8. The performance is slightly worse for the $N = 5$ plot due to the high contrast in $F_2/F_{1,N}$.

This example illustrates several important facts concerning the effectiveness of the Anytime A* algorithms. The error in computing $U^*(\boldsymbol{s})$ and $U(\boldsymbol{s})$ is highly dependent on the $F_{1,N}$ value for slowly permeable obstacles because the trajectory is sometimes very close to these obstacles (introducing additional numerical error). However the trajectories themselves recovered from the methods are very similar between cases A and B, which suggests that our characterization of 'quality' in this example is rather pessimistic. In fact, it is simply the cost that differs (due to F_1) along these trajectories. Because $U(\boldsymbol{s})$ and $U^*(\boldsymbol{s})$ are based on the discretized equations, the error grows substantially as $F_{1,N}$ decreases, but the trajectories remain largely the same. We conducted additional tests on mazes and other periodic/random checkerboards. We found that the anytime methods produced poor results when the obstacles in these examples were slowly permeable and $F_1 \ll F_2$.

3.4.4 Satellite image

In this section we define the speed f through the satellite image in Figure 3.10A. The grayscale intensity of the $m^2 = 350^2$ pixel image directly defines the speed $f \in [0.001, 1.001]$, and we set $\boldsymbol{s} = (337h, 161h)$ and $\boldsymbol{t} = (16h, 188h)$. The interested reader can find the full implementation details and history of this example in [18].

In [18, 57, 58, 59], a *much* more expensive (and thus accurate) heuristic was used that (clearly) accepted a smaller region, whereas here we use the cheap-but-weak φ_0. Using this heuristic, SA*-FMM is only able to restrict small portions of the FMM-accepted domain. Figure 3.10B shows the solution produced by FMM, where color indicates the SA*-FMM-restricted solution.

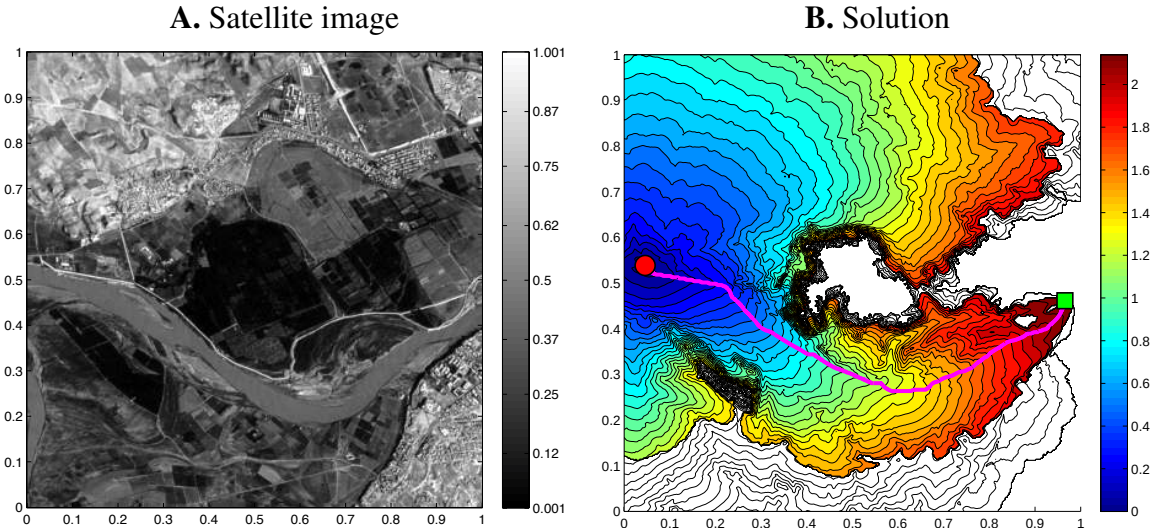


Figure 3.10: **A.** The original satellite image with a colorbar to show the speed directly imported from the grayscale intensity. **B.** The contours of U on the entire domain with the optimal trajectory plotted. The colored contours represent the SA*-FMM-restricted solution, and the FMM solution is shown by both the colored and non-colored contours.

The results in Figure 3.11 verify that the early iterations of ARA*/ANA* quickly produce a solution within 5% error in a fraction of the SA*-FMM runtime. Thus

ARA*/ANA* perform very well on this example in the time up to the SA*-FMM runtime. ARA* terminates shortly after at $16T_{SA}$ seconds, while ANA* takes over twice as long – $43T_{SA}$ seconds. The convergence time greatly differs between the algorithms due to the number of iterations needed: ARA*/ANA* take 33 and 112 iterations, respectively. The later iterations of ANA* incrementally decrease the error starting from an already-small value. E.g., the last 70 ANA* iterations decrease the error from 1% to 0%. This causes the algorithm to take a large number of iterations (and thus time) to produce the final, correct answer. Of course, the effectiveness of ARA* is also highly dependent on the selection of parameters $(w_0, \Delta w)$.

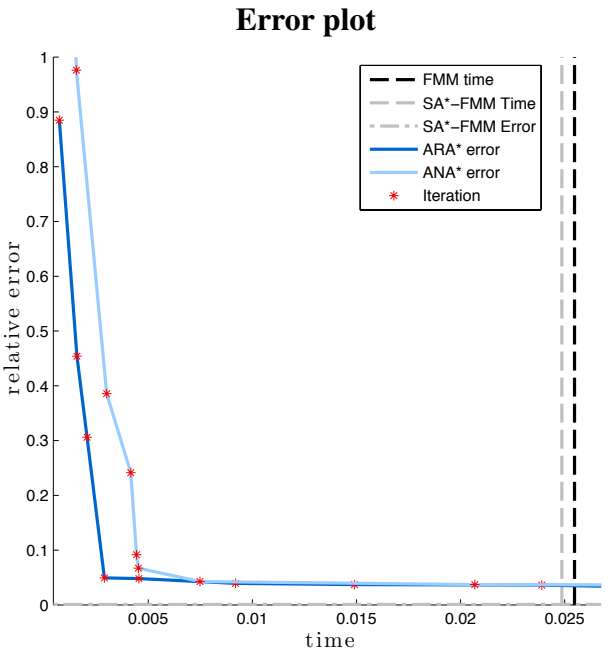


Figure 3.11: Time vs. error of the algorithms. Each * represents a single iteration of the Anytime A* algorithms. The errors plotted correspond to the results found in Table 3.3.

3.4.5 Valleys Example

For this problem we have defined $\boldsymbol{s} = (0.26, 0.76)$, $\boldsymbol{t} = (0.26, 0.24)$, and $m = 501$. The speed function is defined quadratically in terms of the x-distance from the line $x = 0.25$ for $x \geq 0.25$, and defined to be slow inside the obstacles and when $x < 0.25$. We chose our speed function to be

$$f(x,y) = \begin{cases} a & \text{if } (x,y) \in R \\ a + b(x - 0.25)^2 & \text{if } x \geq 0.25 \\ a/2 & \text{if } x < 0.25, \end{cases} \tag{3.16}$$

where $a = 0.01$, $b = 10.0$, and the permeable rectangular obstacles given by R are shown in Figure 3.12A. The solution is shown in figure 3.12B.

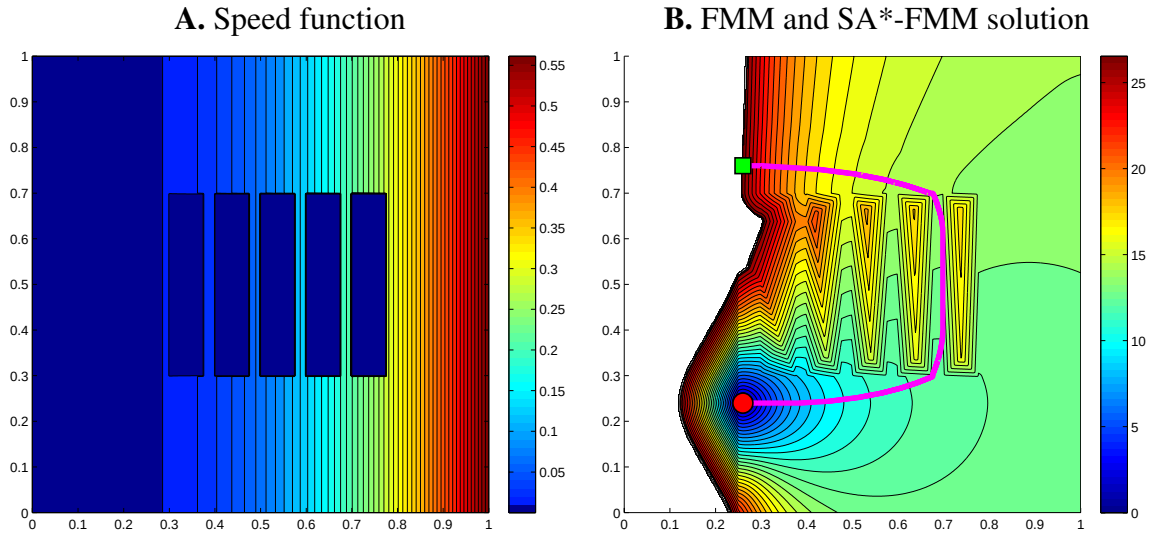


Figure 3.12: **A.** Speed function defined in (3.16), where the speed decreases quadratically as x increases to the right of $x = 0.25$, and there are permeable rectangular obstacles. **B.** The contours of U on the entire domain where $U \leq U(\boldsymbol{s})$. The colored contours represent the SA*-FMM-restricted solution, and the FMM contours are shown by both the colored and non-colored contours. The magenta trajectory represents the optimal trajectory (recovered via FMM).

This section illustrates several important points with this example:

- The class of problems where these algorithms are able to perform well for the purposes of anytime planning are those where there exists several locally optimal trajectories to be discovered as the solution expands outwards. As discussed, the algorithm first aggressively attempts to find a crude solution that is iteratively improved, with the large decrease in suboptimality coming from the trajectory 'jumping' between the various barriers in the domain.
- An interesting phenomenon occurs here since the accuracy of the heuristic φ is low along locally optimal paths due to the large amount of variance in f. This effect occurs when the ratio of $f(\boldsymbol{x})$ to F_2 is very small along any locally optimal path. This is an ideal scenario for the Anytime algorithms since scaling φ by the weighting factor w will help to recover various suboptimal paths by 'pushing' the solution to expand towards the source location along these various paths.

The accuracy of the heuristic is so low that SA* provides almost no restriction, which can be seen in Figure 3.12B since the non-colored contours are barely visible. In fact, the percent of the domain computed by SA*-FMM is 77.6% compared to 78.2% computed by FMM.

The results of the Anytime A*-FMM algorithms can be seen in Figure 3.13. Here we have two different experimental setups for Figures 3.13A&B:

A. The results shown in Figure 3.13A use the default "naïve heuristic" $\varphi = \varphi_0$ with a special choice of ϵ_0 and $\Delta\epsilon$. We had to use slightly inflated values for ϵ and $\Delta\epsilon$ to compensate for the large U-values compared to the small φ values. For the results presented in Figure 3.13A we used the default naïve heuristic φ_0 defined in (3.11) with $\epsilon_0 = F_2/F_1$ and $\Delta\epsilon = \epsilon_0/100$.

B. The results shown in Figure 3.13B use a custom heuristic for φ with the default

choice of $\epsilon_0 = 10$ and $\Delta\epsilon = 0.1$. The custom heuristic used is by integrating the slowness along the manhattan path from $\boldsymbol{x} = (x_1, x_2)$ to $\boldsymbol{s} = (s_1, s_2)$ with only one turn and does not pass through any obstacles. We calculated it to be

$$\varphi(\boldsymbol{x}) = \frac{|x_2 - s_2|}{a + b\,(x_1 - s_1)^2} + \sqrt{\frac{1}{ab}} \tan^{-1}\left(\sqrt{\frac{b}{a}}\,|x_1 - s_1|\right) \quad \text{when } x \geq 0.25 \text{ and outside obstacles.} \tag{3.17}$$

This was computed by computing the time taken in the y-direction (the first term) and adding it to the time taken in the x-direction (the second term). The first term can be computed by distance divided by speed since the speed is constant, and the second term can by computed by integrating the slowness.

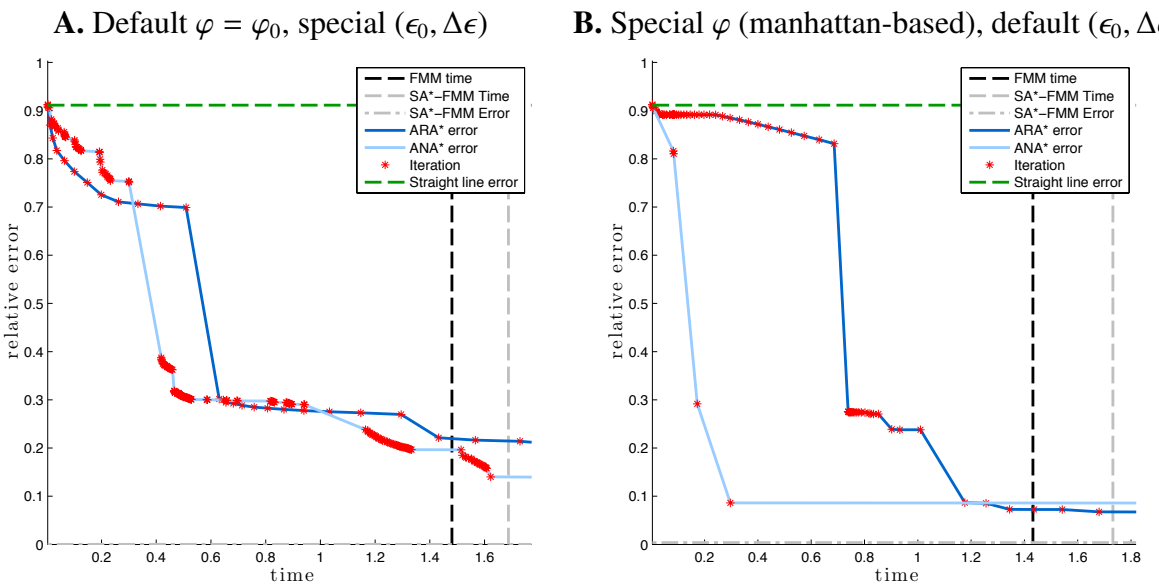


Figure 3.13: Time vs. error of the algorithms. Each * represents a single iteration of the Anytime A* algorithms. The errors plotted correspond to the results found in Table 3.3. **A.** Results using the default "naïve heuristic" $\varphi = \varphi_0$ with a special choice of $(\epsilon_0, \Delta\epsilon) = (F_2/F_1, F_2/F_1/100) \approx (114.5, 1.145)$. **B.** Results using a special heuristic φ defined in (3.17) and the default $(\epsilon_0, \Delta\epsilon) = (10, 0.1)$.

3.4.6 Robotic Arm Example

Finally we consider a robotics example originally presented in [86] where a two-link robotic arm is attempting to move through a space with obstacles. This example illustrates a situation where Anytime A* does not perform well due to the limited number of locally optimal trajectories to be discovered.

- Here the physical space that the arm is moving through is represented by $[0, 1]^2$. We have made a slight modification to the configuration space by making the obstacles smaller to allow for more feasible paths.
- However, the underlying planning space is given by $(\theta_1, \theta_2) \in [0, \pi] \times [0, 2\pi]$ where θ_1 is the angle with the positive x-axis and θ_2 is the angle with the respect to the first link. See Figure 3.14 for an diagram of the arm in the configuration space.
- Due to the unequal size of the domain, but our assumption of equal grid-spacing we discretize the (θ_1, θ_2)-space into an $m \times n = 501 \times 1001$. We discretize the (θ_1, θ_2)-space by a 401×801 grid and test the various algorithms on the problem. Note that by discretizing the (θ_1, θ_2)-space should be considered as an *approximation* to discretizing the actual state space of (θ_1, θ_2) which constitutes a subset of a torus.
- We will allow for self-revolutions with respect to the second link in the arm, which translates to periodic boundary conditions in θ_2 within the Eikonal PDE. Both links in the arm have a length of $R = 0.4$.

  Thus the tip of the robotic arm can be represented by

  $$P = (0.5, 0) + R(\cos\theta_1, \sin(\theta_1)) + R(\cos(\theta_1 + \theta_2 - \pi), \sin(\theta_1 + \theta_2 - \pi)). \quad (3.18)$$

- We choose to define the speed function so that it is faster to move when the arm is in a folded position versus extended, i.e. when θ_2 is further from π the speed will

be faster. Thus

$$f(\theta_1, \theta_2) \;=\; 3 + 2 \cdot \frac{|\theta_2 - \pi|}{\pi}. \tag{3.19}$$

- The initial and final positions of the arm are given in (θ_1, θ_2)-space by $\boldsymbol{s} = (7\pi/8, 5\pi/8)$ and $\boldsymbol{t} = (\pi/8, 11\pi/8)$

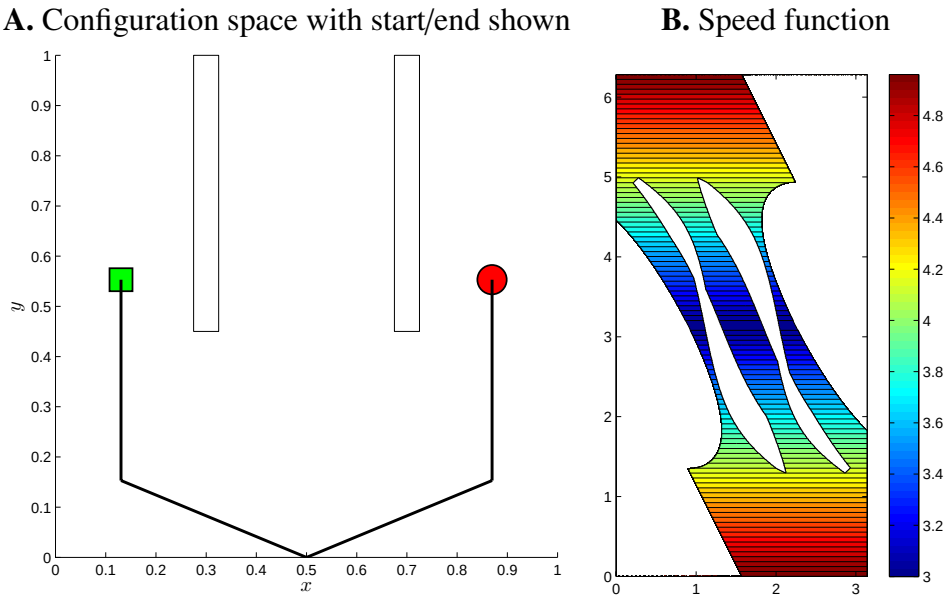


Figure 3.14: **A.** The initial and final configuration of the two-link arm along with the impermeable obstacles. **B.** Speed function defined in (3.19) in (θ_1, θ_2)-space.

Figure 3.15A shows the results of running SA*-FMM and FMM on the problem in the (θ_1, θ_2)-space. The optimal trajectory recovered via FMM is shown in magenta, and the trajectory recovered via SA*-FMM is shown in green. The SA*-FMM trajectory differs significantly from the FMM trajectory due to the additional error committed by the SA*.

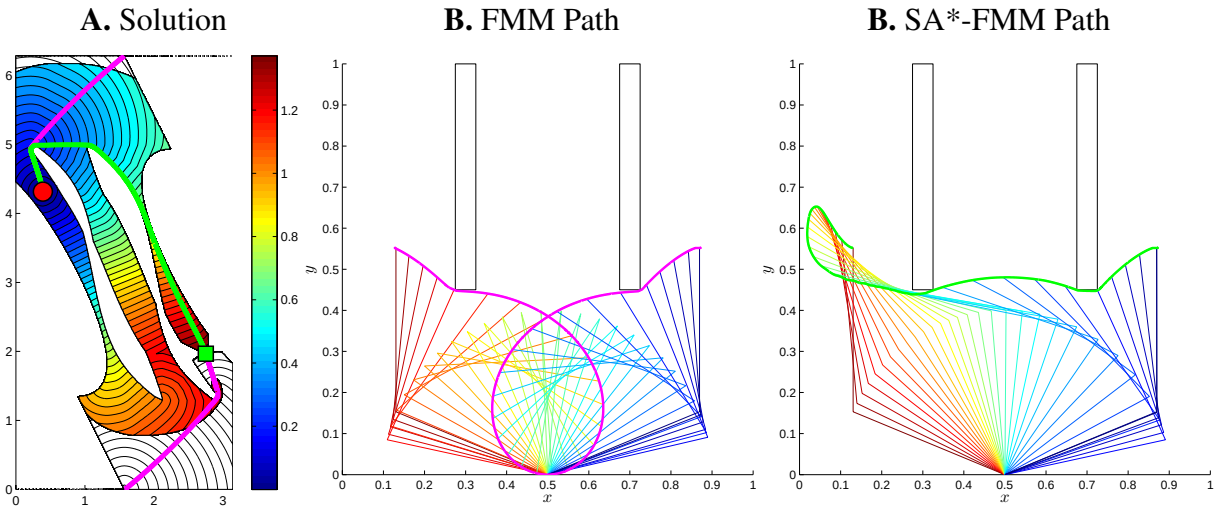


Figure 3.15: **A.** The contours of U on the entire domain where $U \leq U(\boldsymbol{s})$. The colored contours represent the SA*-FMM-restricted solution, and the FMM contours are shown by both the colored and non-colored contours. The magenta trajectory represents the optimal trajectory (recovered via FMM), and the green trajectory represents a locally optimal trajectory recovered via SA*-FMM. **B & C.** Visualization of the two-link arm traveling through the configuration space where red is high-cost and blue is low-cost. The curves shown in purple & green represent the tip of the two-link arm computed by taking the magenta (optimal, FMM-computed) & green (sub-optimal, SA*-FMM-computed) trajectories from **A** and computing the tip location via (3.18).

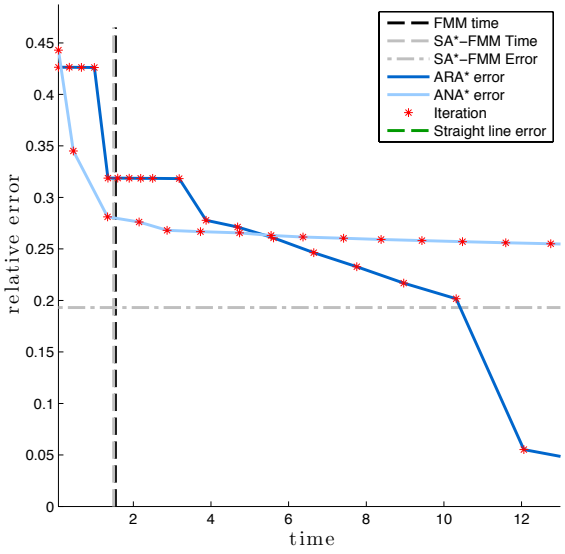
Note that the resulting two paths shown in Figure 3.15A are visualized in the (θ_1, θ_2)-planning-space. Alternatively these paths can be represented in the configuration space where the robotic arm is moving. We have taken the magenta and green paths in Figure 3.15A and computed the location of the arm along the paths and the results are shown in Figures 3.15B and 3.15C, respectively.

The visualization presented in Figure 3.15A shows that there are only four different general routes for trajectories from $\boldsymbol{s}$ to $\boldsymbol{t}$ to travel along. In fact, the anytime algorithms are both able to recover three of these routes as they run, only missing out on the "middle route" when $\theta_2 \approx 2\pi - 2\theta_1$. The presence of so few locally optimal trajectories indicates that the anytime algorithms may not perform well on this problem since they perform

best when there are many locally optimal trajectories to be found.

The results of running ARA* and ANA* on this problem is shown in Figure 3.16. For both results in this Figures 3.16A&B we only show the results until approximately 15 seconds and 1.6 seconds, respectively. Note that both algorithms do converge to the correct solution in finite time, however we only plot the results until 13 seconds and 1.6 seconds in Figures 3.16A and 3.16B, respectively.

A. Error plot until ≈ 13 seconds



B. (Zoom) Error plot until ≈ 1.6 seconds

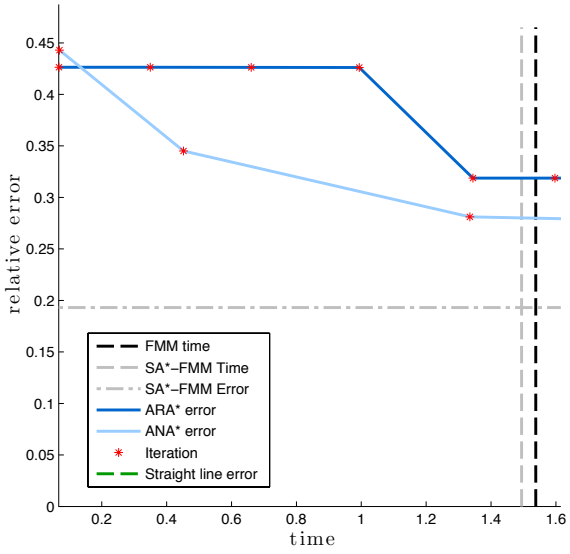


Figure 3.16: Time vs. error of the algorithms. Each * represents a single iteration of the Anytime A* algorithms. The errors plotted correspond to the results found in Table 3.3.

3.5 Conclusions

We have presented a new extension of Anytime A* algorithms from graphs to continuous optimal path planning. Previously these algorithms had only been studied in the discrete setting, where a 'path' travels from node-to-node until reaching the exit-set [30, 31, 46, 7]. In the continuous setting a mesh or grid is used to discretize the do-

main and the value function is recovered at each meshpoint. However, this increases the underlying complexity of the problem and associated algorithms for a number of reasons including an increase on the connectivity of the discretization as well as a more computationally-intense update requirement of each meshpoint. This can introduce non-trivial effects that must be carefully studied and managed.

This is done from the viewpoint of the optimal control and the Eikonal PDE (3.1). The goal of the algorithms is to quickly produce a suboptimal trajectories that can be used for planning applications, and the solution quality improves as runtime is allowed to continue. On problems where SA* and AA* variants of FMM [18, 84, 85] are ineffective at producing useful levels of domain restriction, the Anytime A* algorithms or WA*-FMM could be used to quickly produce some approximate solution.

These Anytime A*-FMM adaptations typically perform best when there are many locally suboptimal trajectories and/or when the heuristic φ severely understimates the time-to-source function V. However, for problems that generally don't satisfy these criteria we found the performance of the anytime algorithms to be subpar to the A*-FMM adaptations; i.e. the solution quality was far worse and did not substantially improve before A*-FMM terminated. We illustrated the performance of these algorithms on many numerical tests presented in §3.4.

There are a number of extensions and potential enhancements to these methods:

- In a high-dimensional space it does not make sense the allocate the entire domain into memory but to rather dynamically build the grid as new nodes are needed; see [46] where this was performed on graphs. We hope to exploit this technique for future testing in higher dimensional state spaces.
- A somewhat different but related issue concerns whether or not to 'restart' the

search after every iteration of Anytime A*-FMM. On graphs it has been claimed that this is a useful procedure when there are many locally optimal trajectories [64]. This could again prove useful on high-dimensional problems where memory allocation could become an issue.

- We attempted to implement label-correcting versions of the Anytime A* algorithms but found the results to be very unappealing, however several improvements such as bounding the number of times a node can be considered on each iteration can be made. A related problem to be studied would be label-correcting versions of Anytime A* algorithms where each node is only allowed to be reconsidered some fixed number of times per iteration.

- The properties of φ that produce desirable results for the anytime algorithms are when the problem itself has many locally optimal trajectories and when φ greatly underestimates V (the minimum time to $\boldsymbol{s}$). Further studies on the effect of φ on the performance of SA*, WA*, and the anytime algorithms (ARA*, ANA*) presented here should be performed, as well as development of alternative heuristics to those commonly used (see [18] for an overview).

- For WA* on graphs (with weight w) a suboptimality bound of $U_{WA}(\boldsymbol{x}) \leq wU_{Dijkstra}(\boldsymbol{x})$ is available (when φ is consistent) at a node $\boldsymbol{x}$, where U_{WA} is the WA*-found solution and $U_{Dijkstra}$ is the Dijkstra's-found solution. We hope to prove a similar suboptimality bound for WA*-FMM on triangulated meshes.

- In this paper we have presented one technique for adjusting the parameter w for the ARA* algorithm based on the current information available as well as stated several ways to choose the parameters. However, more work should be done to analyze what parameters work well with the Anytime A* algorithms (both on graphs and for continuous problems). For example, adjusting ϵ as the ARA* algorithm runs is performed either by reseting to a lower value via (3.13) or decreasing by

a constant Δw at the end of every iteration. However, there are a myriad of ways to choose the next value of ϵ for ARA* to use and more work should be done to study how to select the values used for all parameters.

To the best of our knowledge, there has been no prior work done on adapting Anytime A* algorithms to continuous problems through the use of FMM. We hope that practitioners find our algorithm useful, especially for problems where previous methods fail or are unable to realistically compute solutions. In particular, problems in high-dimensional state spaces suffer from the curse of dimensionality and we hope that aggressive 'anytime' searches can help produce useful suboptimal solutions.

Acknowledgements. The author is grateful to Jur van den Berg for originally suggesting a variant of this project. Thank you to Alexander Vladimirsky for many proofreads and helpful discussions.

CHAPTER 4

A BI-CRITERIA PATH-PLANNING ALGORITHM FOR ROBOTICS APPLICATIONS

4.1 Introduction

The shortest path problem on graphs is among the most studied problems of computer science, with numerous applications ranging from robotics to image registration. Given the node-transition costs, the goal is to find the path minimizing the cumulative cost from a starting position up to a goal state. When there is a single (nonnegative) cost this can be accomplished efficiently by Dijkstra's algorithm [22] and a variety of related *label-setting methods*.

However, in many realistic applications paths must be evaluated and optimized according to several criteria simultaneously (time, cumulative risk, fuel efficiency, etc). Multiple criteria lead to a natural generalization of path optimality: a path is said to be (weakly) *Pareto-optimal* if it cannot be improved according to all criteria simultaneously. For evaluation purposes, each Pareto-optimal path can be represented by a single point, whose coordinates are cumulative costs with respect to each criterion. A collection of such points is called the *Pareto Front* (PF), which is often used by decision makers in a posteriori evaluation of optimal trade-offs. A related practical problem (requiring only a portion of PF) is to optimize with respect to one ("primary") criterion only, but with upper bounds enforced based on the remaining ("secondary") criteria.

It is natural to approach this problem by reducing it to single criterion optimization:

This chapter is based on the paper *A Bi-criteria Path-Planning Algorithm for Robotics Applications* by Z. Clawson, D. Ding, B. Englot, T.A. Frewen, W.M. Sisson, and A. Vladimirsky, and was submitted to IEEE Transactions on Automation Science and Engineering on January 9, 2017.

a new transition-cost is defined as a weighted average of transition penalties specified by all criteria, and Dijkstra's method is then used to recover the "shortest" path based on this new averaged criterion. It is easy to show that the resulting path is always Pareto-optimal for every choice of weights used in the averaging. Unfortunately, this *scalarization approach* has a significant drawback [20]: it finds the paths corresponding to convex parts of PF only, while the non-convexity of PF is quite important in many robotics applications. The positive and negative algorithmic features of scalarization are further discussed in §4.4.4.

The same problem applied to a stochastic setting in the context of a Markov Decision Processes (MDP) is called stochastic shortest path problem, which can be seen as a generalization of the deterministic version considered in this paper. In this case multiple costs can be considered such that a primary cost is optimized, and an arbitrary number of additional costs can be used as constraints. Such problem can generally be solved as a Constrained MDP (CMDP) [1], and recent work to extend this approach is "Chance-constrained" CMDP [54] that works well for large state spaces [36, 24] for robotics applications. However, by considering the additional costs as constraints, the PF cannot be extracted.

Numerous generalizations of label-setting methods have also been developed to recover *all* Pareto optimal paths [37, 76, 82, 49, 50, 73, 47, 48]. However, all of these algorithms were designed for general graphs, and their efficiency advantages are less obvious for highly-refined geometrically embedded graphs, where Pareto-optimal paths are particularly plentiful. (This is precisely the case for meshes or random-sampling based graphs intended to approximate motion planning in continuous domains.) In this paper we describe a simple method for bi-criteria path planning on such graphs, with an efficient approach for approximating the entire PF. The key idea is based on keeping

track of the "budget" remaining to satisfy the constraints based on the secondary criterion. This approach was previously used in continuous optimal control by Kumar and Vladimirsky [45] to approximate the discontinuous value function on the augmented state space. More recently, this technique was extended to hybrid systems modeling constraints on reset-renewable resources [79]. In the discrete setting, the same approach was employed as one of the modules in hierarchical multi-objective planners [23]; our current paper extends that earlier conference publication.

The key components of our method are discussed in §4.2, followed by the implementation notes in §4.3. In §4.4 we provide the results of numerical tests on Probabilistic RoadMap graphs (PRM) in two-dimensional domains with complex geometry. Our algorithms were also implemented on a realistic heterogeneous robotic system and tested in field experiments described in §4.5. Finally, open problems and directions for future work are covered in §4.6.

4.2 The Augmented State Space Approach

Consider a directed graph $\mathcal{G}$ on the set of nodes $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{x}_{n+1} = \boldsymbol{s}\}$ and edges E between nodes. We will choose $\boldsymbol{s}$ to be a special "source" node (the robot's current position) and our goal will be to find optimal paths starting from $\boldsymbol{s}$. For convenience, we will also use the notation $\mathcal{N}(\boldsymbol{x}_j) = \mathcal{N}_j$ for the set of all nodes $\boldsymbol{x}_i$ from which there exists a direct transition to $\boldsymbol{x}_j$. We will assume that the cost of each such transition C_{ij} is positive. We begin with a quick review of the standard methods for the classical shortest path problems.

4.2.1 The single criterion case

Using $\Phi(P)$ to denote the cumulative cost (i.e., the sum of transition costs C_{ij}'s) along a path P, the usual goal is to minimize Φ over $\mathcal{P}_j$, the set of all paths from $\boldsymbol{s}$ to $\boldsymbol{x}_j \in X$. The standard dynamic programming approach is to introduce the *value function* $U(\boldsymbol{x}_j) = U_j$, describing the total cost along any such optimal path. Bellman's optimality principle yields the usual coupled system of equations:

$$U_j \;=\; \min_{\boldsymbol{x}_i \in \mathcal{N}_j} \left\{ C_{ij} \;+\; U_i \right\}, \qquad \forall\; j = 1, 2, \ldots, n \tag{4.1}$$

with $U_{n+1} = U(\boldsymbol{s}) = 0$. Throughout the paper we will take the minimum over an empty set to be $+\infty$. The vector $U = (U_1, \ldots, U_n)$ can be formally viewed as a fixed point of an operator $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^n$ defined componentwise by (4.1). A naive approach to solving this system is to use *value iteration*: starting with an overestimate initial guess $u_0 \in \mathbb{R}^n$, we could repeatedly apply $\mathcal{T}$, obtaining the correct solution vector U in at most n iterations. This results in $O(\kappa n^2)$ computational cost as long as the in-degrees of all nodes are bounded: $\left|\mathcal{N}_j\right| < \kappa$ for some constant $\kappa \ll n$. The process can be further accelerated using the Gauss-Seidel relaxation, but then the number of iterations will strongly depend on the ordering of the nodes within each iteration. For acyclic graphs, a standard topological ordering of nodes can be used to essentially decouple the system of equations. In this case, the Gauss-Seidel relaxation converges after a single iteration, yielding the $O(n)$ computational complexity. For a general directed graph, such a *causal* ordering of nodes is a priori unknown, but can be recovered at runtime by exploiting the monotonicity of U values along every optimal path. This is the essential idea behind Dijkstra's classical algorithm [22], which relies on heap data structures and solves this system in $O(n \log n)$ operations. Alternatively, a class of *label-correcting* algorithms (e.g., [9, 10, 8]) attempt to approximate the same causal ordering, while avoiding the use of heap-sort data structures. These algorithms still have the $O(n^2)$

worst-case complexity, but in practice are known to be at least as efficient as Dijkstra's on many types of graphs [9].

When we are interested in optimal directions from $\boldsymbol{s}$ to a specific node $\boldsymbol{t} \in X$ only, Dijkstra's method can be terminated as soon as $U(\boldsymbol{t})$ is computed. (Note: Dijkstra's algorithm typically expands computations outward from $\boldsymbol{s}$.) An A* method [32] can be viewed as a speed-up technique, which further restricts the computational domain (to a neighborhood of $(\boldsymbol{s}, \boldsymbol{t})$ optimal path) using heuristic underestimates of the cost-to-go function. More recent extensions include "Anytime A*" (to ensure early availability of good suboptimal paths) [30, 46, 7] and a number of algorithms for dynamic environments (where some of the C_{ij} values might change before we reach $\boldsymbol{t}$) [74, 75, 43].

4.2.2 Bi-criteria optimal path planning: different value functions and their DP equations

We will now assume that an alternative criterion for evaluating path quality is defined by specifying "secondary costs" $c_{ij} > 0$ for all the transitions in this graph. Similarly, we define $\phi(P)$ to be the cumulative secondary cost (based on c_{ij}'s) along P. In this setting, it is useful to consider several different value functions. The secondary (unconstrained) value function is $V_j = \min_{P \in \mathcal{P}_j} \phi(P)$. It satisfies a similar system of equations: $V_{\boldsymbol{s}} = 0$ and

$$V_j = \min_{\boldsymbol{x}_i \in \mathcal{N}_j} \left\{ c_{ij} + V_i \right\}, \quad j \leq n.$$

A natural question to ask is how much cost must be incurred to traverse a path selected to optimize a *different* criterion. We define $\tilde{V}_j$ as $\phi(P)$ minimized over the set of all primary-optimal paths $\left\{ P \in \mathcal{P}_j \mid \Phi(P) = U_j \right\}$. Thus, $\tilde{V}_{\boldsymbol{s}} = 0$, and if we define

$\mathcal{N}'_j = \underset{\boldsymbol{x}_i \in \mathcal{N}_j}{\operatorname{argmin}} \left\{ C_{ij} + U_i \right\} \subset \mathcal{N}_j$, then

$$\tilde{V}_j = \min_{\boldsymbol{x}_i \in \mathcal{N}'_j} \left\{ c_{ij} + \tilde{V}_i \right\}, \quad j \leq n.$$

Similarly, we define $\tilde{U}_j$ as $\Phi(P)$ minimized over the set of all secondary-optimal paths $\left\{ P \in \mathcal{P}_j \mid \phi(P) = V_j \right\}$. Thus, $\tilde{U}_s = 0$, and if we define $\mathcal{N}''_j = \underset{\boldsymbol{x}_i \in \mathcal{N}_j}{\operatorname{argmin}} \left\{ c_{ij} + V_i \right\} \subset \mathcal{N}_j$, then

$$\tilde{U}_j = \min_{\boldsymbol{x}_i \in \mathcal{N}''_j} \left\{ C_{ij} + \tilde{U}_i \right\}, \quad j \leq n.$$

All U_j's can be efficiently obtained by the standard Dijkstra's method with $\tilde{V}_j$'s computed in the process as well. The same is true for V_j's and $\tilde{U}_j$'s.

Returning to our main goal, we now define the primary-constrained-by-secondary value function $W(\boldsymbol{x}_j, b) = W_j^b$ as $\Phi(P)$ (the accumulated primary cost) minimized over $\left\{ P \in \mathcal{P}_j \mid \phi(P) \leq b \right\}$ (the paths with the cumulative secondary cost not exceeding b). We will say that b is *the remaining budget* to satisfy the secondary cost constraint. This definition and the positivity of secondary costs yield several useful properties of W:

1. W_j^b is a monotone non-increasing function of b.
2. $b < V_j \iff W_j^b = +\infty$.
3. $b = V_j \iff W_j^b = \tilde{U}_j$.
4. $b \geq \tilde{V}_j \iff W_j^b = U_j$.

We will use the notation

$$\beta(i, b, j) \;=\; b \;-\; c_{ij} \tag{4.2}$$

to define the set of feasible transitions on level b:

$$\mathcal{N}_j(b) = \{ \boldsymbol{x}_i \in \mathcal{N}_j \mid \beta(i, b, j) \geq 0 \}.$$

The dynamic programming equations for W are then $W_s^b = 0$ for all b and

$$W_j^b = \min_{\boldsymbol{x}_i \in \mathcal{N}_j(b)} \left\{ C_{ij} + W_i^{\beta(i,b,j)} \right\}, \qquad j \le n,\ b > 0. \tag{4.3}$$

For notational simplicity, it will be sometimes useful to have W defined for non-positive budgets; in such cases we will assume $W_j^b = +\infty$ whenever $b \le 0$ and $j \le n$. The Pareto Front for each node $\boldsymbol{x}_j$ is the set of pairs $\left(b, W_j^b\right)$, where a new reduction in the primary cost is achieved:

$$\mathcal{PF}_j = \left\{ \left(b, W_j^b\right) \mid W_j^b < W_j^{b'},\ \forall\, b' < b \right\}. \tag{4.4}$$

For each fixed $\boldsymbol{x}_j$, the value function W_j^b is piecewise-constant on b-intervals and the entries in $\mathcal{PF}_j$ provide the boundaries/values for these intervals.

Given the above properties of W, it is only necessary to solve the system (4.3) for $b \in [0, B]$, where B is the *maximal budget level*. E.g., for many applications it might be known a priori that a secondary cumulative cost above some B is unacceptable. On the other hand, if the goal is to recover the entire $\mathcal{PF}_j$ for a specific $\boldsymbol{x}_j \in X$, we can choose $B = \tilde{V}_j$, or even $B = \max_i \tilde{V}_i$ to accomplish this for all $\boldsymbol{x}_j$.

Since all c_{ij} are positive, we observe that the system (4.3) is *explicitly causal*: the expanded graph on the nodes $\left\{ \boldsymbol{x}_j^b \mid \boldsymbol{x}_j \in X, b \ge 0 \right\}$ is acyclic and the causal ordering of the nodes is available a priori. The system (4.3) can be solved by a single Gauss-Seidel sweep in the direction of increasing b.

4.2.3 The basic upward-sweep algorithm

For simplicity, we will first assume that all secondary costs are positive and quantized:

$$\exists\, \delta > 0 \ \text{ s.t. all } c_{ij} \in \{\delta, 2\delta, 3\delta, \ldots\}.$$

This also implies that W_j^b is only defined for $b \in \mathcal{B} = \{0, \delta, 2\delta, \ldots, B := m\delta\}$. A simple implementation (Algorithm 6, described below) takes advantage of this structure by storing W_j^b as an array of values for each node $\boldsymbol{x}_j \in X \setminus \{\boldsymbol{s}\}$ and $b \in \mathcal{B}$. For $\boldsymbol{s}$ we will need only one value $W_s = W_s^b = 0$ for all $b \in \mathcal{B}$. For each remaining node $\boldsymbol{x}_j \in X \setminus \{\boldsymbol{s}\}$ with budget $b \in \mathcal{B}$ we can compute the primary-constrained-by-secondary value function W_j^b using formula (4.3).

Algorithm 6: The basic explicitly causal (single-sweep) algorithm.

Initialization:

Compute $U, V, \tilde{U}, \tilde{V}$ for all nodes by Dijkstra's method.

Set $W_s := 0$

Main Loop:

foreach $b = \delta, \ldots, B$ **do**
    foreach $\boldsymbol{x}_j \in X \setminus \{\boldsymbol{s}\}$ **do**
        if *(*$U_j < \infty$*) AND (*$b \geq V_j$*)* **then**
            if $b = V_j$ **then**
                $W_j^b := \tilde{U}_j$;
            else
                if $b < \tilde{V}_j$ **then**
                    Compute W_j^b from equation (4.3);
                else
                    $W_j^b := U_j$;
        else
            $W_j^b := +\infty$.

The explicit causality present in the system is taken advantage of by the algorithm, leading to at most $|X| \cdot |\mathcal{B}| = nm$ function calls to solve equation (4.3). This results in an appealing $O(nm)$ complexity, linear in the number of nodes n and the number of discrete budget levels m.

4.2.4 Quantizing secondary costs and approximating $\mathcal{PF}$

In the general case, the secondary costs need not be quantized, and even if they are, the resulting number of budget levels m might be prohibitively high for online path-planning. But a slight generalization of Algorithm 6 is still applicable to produce a conservative approximation of W and $\mathcal{PF}$. This approach relies on "quantization by overestimation" of secondary edge weights with a chosen $\delta > 0$:

$$\widehat{c_{ij}} \;:=\; \delta\left\lceil \frac{c_{ij}}{\delta} \right\rceil \;\geq\; c_{ij}. \tag{4.5}$$

Similarly, we can define $\hat{\phi}(P)$ to be the cumulative quantized secondary cost along P.

The definition of β in (4.2) is then naturally modified to

$$\beta(i, b, j) \;=\; b \,-\, \widehat{c_{ij}}, \tag{4.6}$$

with the set of feasible transitions on level $b \in \mathcal{B}$ still defined as $N_j(b) = \{\boldsymbol{x}_i \in N_j \mid \beta(i, b, j) \geq 0\}$. Modulo these changes, the new value function $\widehat{W}_j^b$ is also defined by (4.3) for all $b \in \mathcal{B}$. It can be similarly computed by Algorithm 6 if the condition on line 7 is replaced by

$$\texttt{if } b \in [V_j,\, V_j + \delta) \texttt{ then ...}$$

The above modifications ensure that $\hat{\phi} \geq \phi$ for every path and $W_j^b \leq \widehat{W}_j^b$ for all $b \in \mathcal{B}$, $\boldsymbol{x}_j \in X$. Moreover, if $\widehat{W}_j^b$ is finite, there always exists some "optimal path" $P \in \mathcal{P}_j$ such that $\hat{\phi}(P) \leq b$ and $\Phi(P) = \widehat{W}_j^b$. Of course, there is no guarantee that such a path P is truly Pareto-optimal with respect to the real (non-quantized) c_{ij} values, but $\hat{\phi}(P) \to \phi(P)$ and the obtained $\widehat{\mathcal{PF}}_j$ converges to the non-quantized Pareto Front as $\delta \to 0$. In fact, it is not hard to obtain the bound on $\hat{\phi}(P) - \phi(P)$ by using the upper bound on the number of transitions in Pareto-optimal paths. Let $c = \min_{i,j} c_{ij}$ and $C = \min_{i,j} C_{ij}$. If $k(P)$ is the

number of transitions on some Pareto-optimal path P from $\boldsymbol{s}$ to $\boldsymbol{x}_j$, then

$$k(P) \;\leq\; K \;:=\; \min\left(\frac{\tilde{V}_j}{c},\; \frac{\tilde{U}_j}{C}\right).$$

Since Algorithm 6 overestimates the secondary cost of each transition by at most δ, we know that

$$\phi(P) \;\geq\; \hat{\phi}(P) \;-\; K\delta.$$

In the following sections we show that much more accurate "budget slackness" estimates can be also produced at runtime.

4.3 Implementation Details

A discrete multi-objective optimization problem is defined by the choice of domain Ω, the graph $\mathcal{G}$ encoding allowable (collision-free) transitions in Ω, and the primary and secondary transition cost functions C and c defined on the edges of that graph. Our application area is path planning for a ground robot traveling toward a goal waypoint $\boldsymbol{x}_j \in X$ in the presence of enemy threat(s). The Pareto Front consists of pairs of threat-exposure/distance values (each pair corresponding to a different path), where any decrease in threat exposure results in an increase in distance traveled and vice-versa.

While recovering the Pareto Front is an important task, deciding how to use it in a realistic path-planning application is equally important. If c is based on the threat exposure and the maximal allowable exposure $B \geq V_j$ is specified in advance, the entire $\mathcal{PF}_j$ is not needed. Instead, the algorithm can be used in a fully automatic regime, selecting the path corresponding to W_j^B. Alternatively, choosing $B = \tilde{V}_j$ we can recover the entire Pareto Front, and a human expert can then analyze it to select the best trade-off between Φ and ϕ. This is the setting used in our field experiments described in Section 4.5.

4.3.1 Discretization parameter δ.

After the designation of Ω, C, and c, the only remaining parameter needed for Algorithm 1 is $\delta > 0$. For a fixed graph $\mathcal{G}$, the choice of δ strongly influences the performance:

1. The selection of δ provides direct control over the runtime of the algorithm on $\mathcal{G}$. Due to the fact that our robot's sampling-based planning process produces graphs of a predictable size, a fixed number of secondary budget levels m ensures a predictable amount of computation time. Since the complexity is linear in m, the user can use the largest affordable m to define

$$\delta \;:=\; \tilde{V}_j \,/\, m, \tag{4.7}$$

   where $\tilde{V}_j$ is based on the *non-quantized* secondary weights c.

2. The size of δ also controls the coarseness of the approximate Pareto Front. After all, $\left|\widehat{\mathcal{PF}}_j\right| \leq m = B/\delta$, and the approximation is guaranteed to be very coarse if the number of Pareto-optimal paths in $\mathcal{P}_j$ is significantly higher.

3. Even if m is large enough to ensure that the number of Pareto optimal paths is captured correctly, there may be still an error in approximating $\mathcal{PF}_j$ due to quantization of secondary costs. For each pair $(b, \widehat{W}_j^b) \in \widehat{\mathcal{PF}}_j$ there is a corresponding path $P_j^b \in \mathcal{PF}_j$, whose *budget slackness* can be defined as

$$S_j^b \;=\; S\left(P_j^b\right) \;=\; \hat{\phi}\left(P_j^b\right) - \phi\left(P_j^b\right) \;=\; b - \phi\left(P_b\right) \;\geq\; 0, \tag{4.8}$$

   which can be considered a post-processing technique. Alternatively, if the last transition in this path is from $\boldsymbol{x}_i$ to $\boldsymbol{x}_j$, this can be used to recursively define the slackness measurement

$$S_j^b \;=\; \widehat{c}_{ij} - c_{ij} + S_i^{b-\widehat{c}_{ij}},$$

and compute it simultaneously with $\widehat{W}_j^b$ as an error estimate for $\widehat{\mathcal{PF}}_j$. This approach was introduced in [65].

We point out two possible algorithmic improvements not used in our current implementation:

1. As described, the memory footprint of our algorithm is proportional to nB/δ since all W values are stored on every budget level. In principle, we need to store only finite values larger than U_j; for each node $\boldsymbol{x}_j$, their number is $m(j) = (\tilde{V}_j - V_j)/\delta$. This would entail obvious modifications of the main loop of Algorithm 6 since we would only need to update the "still constrained" nodes:

$$\text{Still\_Constrained}(b) = \left\{\boldsymbol{x}_j \in X\backslash\{\boldsymbol{s}\} \mid b \in \left[V_j, \tilde{V}_j\right)\right\}.$$

   This set can be stored as a linked list and efficiently updated as b increases, especially if all nodes are pre-sorted based on $\tilde{V}_j$'s.

2. If the computational time available only permits for a coarse budget quantization (i.e. δ is large), an initial application of Algorithm 6 may result in failure to recover a suitable number of paths from a set of evenly-spaced budget levels. Large differences in secondary cost often exist between paths that lie in different homotopy classes, and individual homotopy classes may contain a multitude of paths with small differences in secondary cost. To remedy this, δ might be refined adaptively, for a specific range of budget values. The slackness measurements can be employed to determine when such a refinement is necessary.

4.3.2 Non-monotone convergence: a case study.

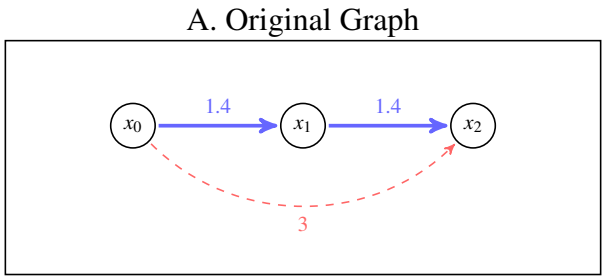
In real-world path planning examples, the true secondary transition costs c_{ij} are typically not quantized, and equation (4.5) introduces a small amount of error dependent on δ. These errors will vanish as $\delta \to 0$, but somewhat counterintuitively the convergence is generally not monotone.

This issue can be illustrated even in single criterion path-planning problems. Consider a simple graph in Fig. 4.1A, where there are two paths to travel from $\boldsymbol{x}_0$ to $\boldsymbol{x}_2$. If the edge-weights for this graph were quantized with formula (4.5), an upper bound for a given path would be:

$$\hat{\phi}(P) \;\leq\; \phi(P) \;+\; \delta \cdot k(P).$$

Figs. 4.1B & 4.1C each show the quantized cost along each path along with this upper bound. There are several interesting features that this example illustrates. First, the convergence of the quantized edge weights is non-monotone, as expected. Further, as $\delta \to 0$, the minimum-cost path corresponding to the quantized edge-weights switches between the two feasible paths in the graph. Fig. 4.2 shows the graph with quantized weights $\widehat{c}$ for specific values of δ, where the path shown in bold is the optimal quantized path. Finally, as δ decreases, a threshold ($\delta = 0.1$) is passed where the quantized cost along the top (truly optimal) path is always less than the quantized cost along the bottom (suboptimal) path.

Returning to the bi-criteria planning, the monotone decrease of errors due to quantization can be ensured if every path P satisfying $\hat{\phi}(P) \leq b$ for a fixed value of δ will still satisfy the same inequality as δ decreases. If we define a sequence $\delta_k = B/m_k$, the simplest sufficient condition is to use $m_{k+1} = 2m_k$. This is the approach employed in all experiments of Section 4.4. For the rest of the paper we suppress the hats on the W's for



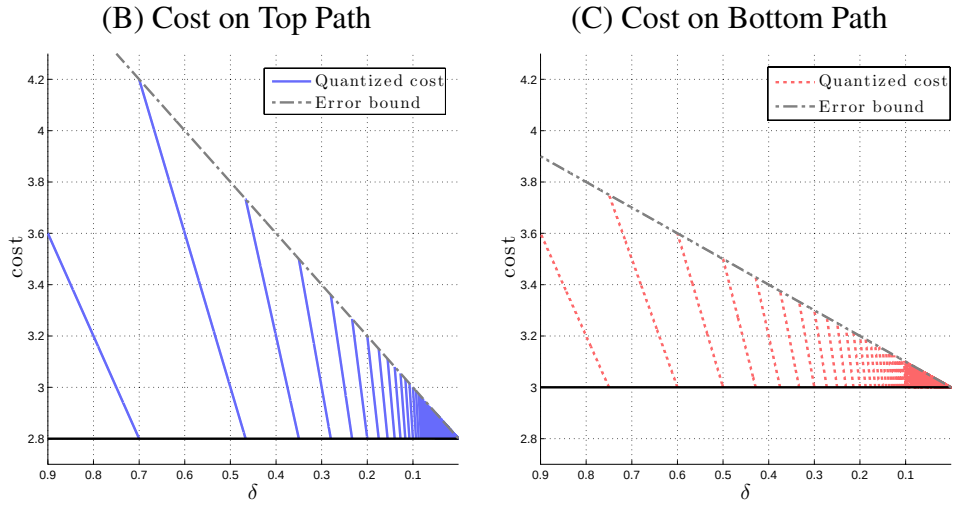


Figure 4.1: (A) Graph with two feasible paths from $\boldsymbol{x}_0$ to $\boldsymbol{x}_2$. (B) & (C) Quantized cost along the top and bottom paths (respectively) as δ decreases (left-to-right). The black lines represent the true cost in each figure (2.8 and 3, respectively).

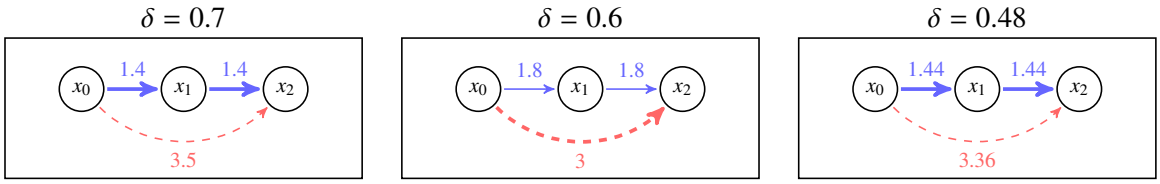


Figure 4.2: The graph in Figure 4.1A with quantized edge-weights for specific values of δ (decreasing from left-to-right). The optimal path in each subfigure is in bold.

the sake of notational convenience; the slackness measurements corresponding to each value of δ are shown for diagnostic purposes in all experiments of Section 4.4.

4.4 Benchmarking on Synthetic Data

We aim to gain further understanding of the proposed algorithm in applications of robotic path planning by focusing most of the attention on a simple scenario. Here, in contrast to Section 5, our goal is to study the convergence properties of the algorithm by refining two accuracy parameters: the number of PRM nodes in the graph and the number of budget levels (determined by δ). Computations are done offline on a laptop rather than a robotic system, allowing for more computational time than in a real scenario. We assume that the occupancy map and threats are known a priori so that we expend no effort in learning this information. All tests are conducted on a laptop with an Intel i7-4712HQ CPU (2.3GHz quad-core) with 16GB of memory.

4.4.1 Case Study Setup

The results presented in this paper require a graph that accurately represents the motion of a robot in an environment with obstacles, where nodes correspond to collision-free configurations and edges correspond to feasible transitions between these configurations. We are particularly interested in graphs that contain a large number of transitions between any two nodes. There are a number of algorithms capable of generating such graphs, such as the Rapidly-exploring Random Graphs (RRG) [40] or Probabilistic Roadmap (PRM) [42] algorithms. In this work we chose to use the PRM algorithm as implemented in Open Motion Planning Library (OMPL) [78] as a baseline algorithm

to generate the roadmap/graph. In essence, the PRM algorithm generates a connected graph representing the robot's state (e.g. position, orientation, etc...) in the environment. The graph is generated by randomly sampling points in the state-space of the robot (called the configuration space in robotics) and trying to connect existing vertices of the graph within the sampled configuration.

In case studies examined, we construct a roadmap as a directed graph $\mathcal{G} = (X, E)$ where X is the set of nodes and E is the set of transitions (edges). The nodes of $\mathcal{G}$ represent the positions of the robot in 2D, i.e. $\boldsymbol{x} \in X \subset \mathbb{R}^2$, and an edge $e = \left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) \in E$, $\boldsymbol{x}_i, \boldsymbol{x}_j \in X$ represents that there is a collision-free line segment from node $\boldsymbol{x}_i$ to node $\boldsymbol{x}_j$ in the PRM graph. Each edge of $\mathcal{G}$ is assigned a primary and secondary transition cost by two cost functions $C, c : E \to \mathbb{R}^+$, respectively.

For the configuration space we chose $\Omega \subset \mathbb{R}^2$ to be sampled by the OMPL planning library [78] with the default uniform (unbiased) sampling, and two slight modifications: we increased the number of points along each edge that are collision-checked for obstacles and removed self-transitions between nodes. For real-world examples the roadmap $\mathcal{G}$ is typically generated by running the PRM algorithm for a fixed amount of time. However, for testing purposes we specify the number of nodes desired in $\mathcal{G}$.

Two test cases are presented:

- Sections 4.4.3–4.4.6 present an example that uses an occupancy map generated by the Hector SLAM algorithm [44] via a LIDAR sensor in a real testing facility.
- Section 4.4.7 presents a more complicated scenario in a synthetic environment to illustrate the much broader scope of the algorithm.

Both tests assume that the physical dimensions of Ω are $450\text{m} \times 450\text{m}$ (so each pixel of the occupancy map is 1m^2). Each edge is generated by connecting vertices in the graph

that are 'close together' via straight-line segments and collision-checking the transition (edge) with a sphere of radius 5m, which represents the physical size of the robot.

The first occupancy map considered is shown in Fig. 4.3A; the color black represents occupied areas (walls) whereas white represents the unoccupied areas (configuration space). The concentric circles represent the contours of a threat level function (to be defined). A PRM-generated roadmap with $2,048$ nodes and $97,164$ edges over this occupancy map is shown in Fig. 4.3B.

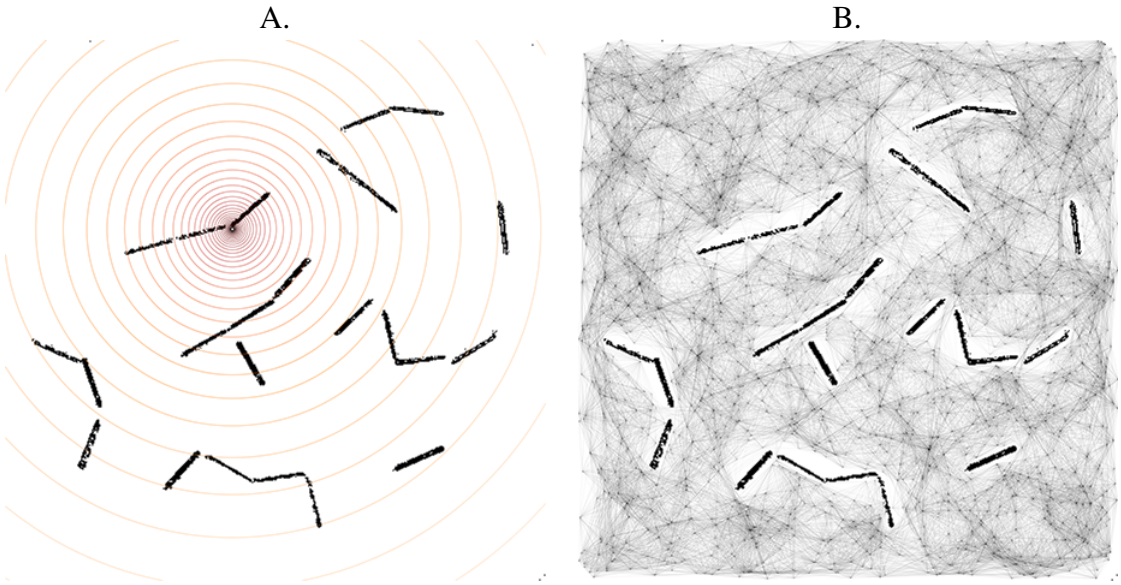


Figure 4.3: (A) Environment with obstacles and contours of a threat exposure function. (B) PRM Roadmap with $2,048$ nodes and $97,164$ edges.

4.4.2 Cost Functions

We consider two cost functions D and T for the multi-objective planning problem. The cost function D represents the Euclidean distance between two adjacent nodes in X, i.e. $D(e) = \left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|$ where $e = (\boldsymbol{x}_i, \boldsymbol{x}_j)$.

The cost function T denotes the threat exposure along an edge in the graph. To

define T we assume that there are threats $\mathcal{K} = \{1, 2, \ldots, N_T\}$ in the environment, which are indicated by their position $\boldsymbol{p}_k \in \mathbb{R}^2$, severity s_k, minimum radius $r_k \geq 0$, and visibility radius $R_k \geq 0$ for each $k \in \mathcal{K}$. For each threat, these parameters define a threat exposure function

$$\tau_k(\boldsymbol{x}) = \begin{cases} s_k / r_k^2 & \text{if } \left\|\boldsymbol{x} - \boldsymbol{p}_k\right\| \leq r_k \\ s_k / \left\|\boldsymbol{x} - \boldsymbol{p}_k\right\|^2 & \text{if } \left\|\boldsymbol{x} - \boldsymbol{p}_k\right\| \in (r_k, R_k) \\ s_k / R_k^2 & \text{if } \left\|\boldsymbol{x} - \boldsymbol{p}_k\right\| \geq R_k \end{cases}, \tag{4.9}$$

for $\boldsymbol{x} \in \mathbb{R}^2$. Fig. 4.3A shows the contours of τ for a single threat with $s = 20$, $r = 5$m, and $R = +\infty$. For simplicity we will always assume that $R_k = +\infty$ for all $k \in \mathcal{K}$ in all examples.

The cumulative threat exposure function is then defined as the integral of the instantaneous threat exposure along the transition (edge), summed over all threats $k \in K$

$$T(e) = \sum_{k \in \mathcal{K}} \int_0^1 \tau_k \left[(1 - t) \cdot \boldsymbol{x}_i + t \cdot \boldsymbol{x}_j\right] \left\|\boldsymbol{x}_j - \boldsymbol{x}_i\right\| \, dt \tag{4.10}$$

where $e = (\boldsymbol{x}_i, \boldsymbol{x}_j) \in E$ is parameterized above by assuming the robot moves at a constant speed. Note that in (4.10), we penalize both the proximity to each threat and the duration of exposure. An example threat placement in the environment is shown in Fig. 4.3A along with the associated contours of the threat exposure function.

4.4.3 Pareto Front

We use the proposed multi-objective planning algorithm to identify paths that lie on the Pareto front of the primary and secondary cost. Our approach enables the re-use of a roadmap for searches under different objectives and constraints.

We rely on the two cost functions D and T described in Section 4.4.2. In every

Pareto Front shown we will reserve the horizontal axis for accumulated values of T and the vertical axis for accumulated values of D. We first point out a Pareto Front in Fig. 4.4A that is generated for the environmental setup shown in Fig. 4.3. The Pareto Front is color-coded (dark blue to magenta) indicating the cumulative threat exposure T along the path (low-to-high), where each color-coded Pareto-optimal path within the configuration space is shown in Fig. 4.4B. As explained in Section 4.3 there is additional error due to the quantization of the secondary costs. As discussed, one method for measuring this error is to calculate the *true secondary cost* along each path, which is shown by the gray markers in every Pareto Front plot.

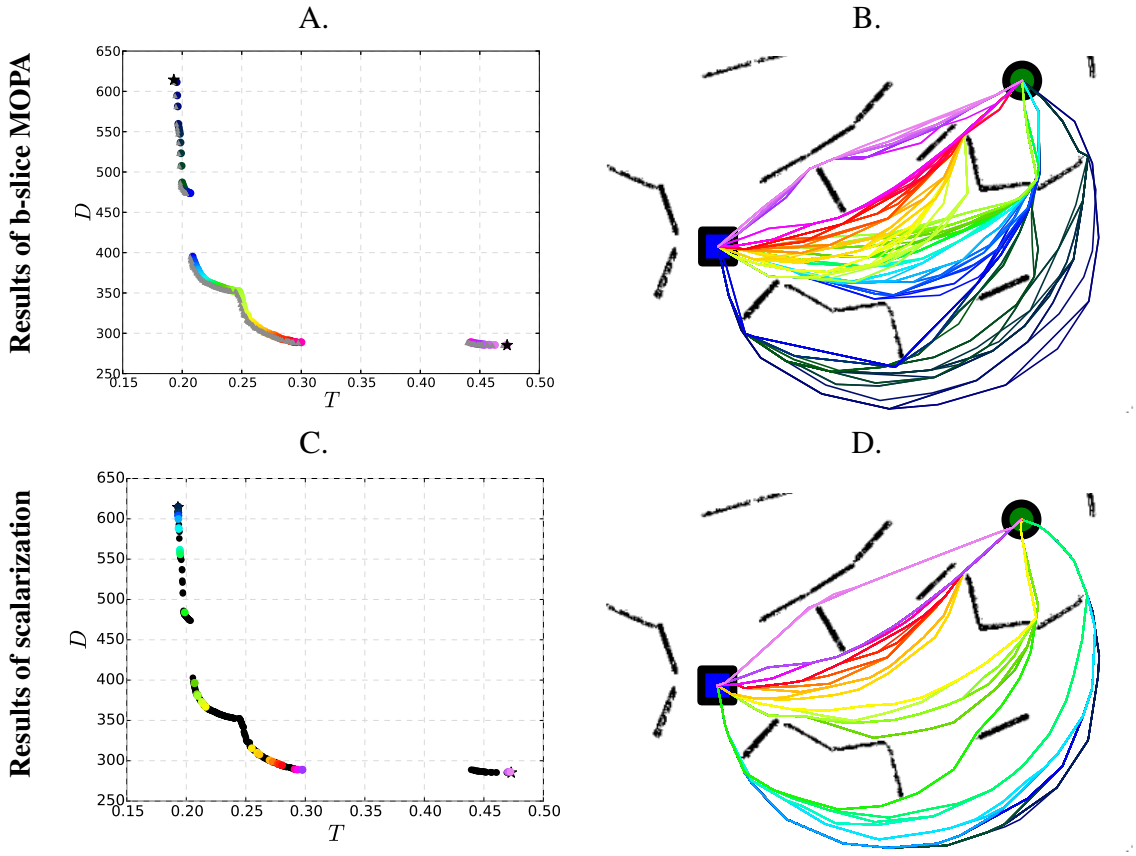


Figure 4.4: Test case with $C = D$ and $c = T$ and the same parameters as Figure 4.3. (A) Pareto Front with a strong non-convexity. The number of budget levels was chosen to be $m = 2048$. (B) Pareto-optimal paths in the configuration space, color-coded to correspond to the Pareto Front in Figure 4.4A. (C) The Pareto Front produced by scalarization algorithm is shown in color. For comparison the gray curve from Figure 4.4A is also shown here in black. (D) Pareto-optimal paths in the configuration space, color-coded to correspond with the Pareto Front in Figure 4.4C.

4.4.4 Comparison with scalarization

The scalarization algorithm operates by minimizing a weighted sum of the objectives to recover Pareto-optimal solutions. In the context of bi-criteria path-planning on graphs, this means that we select some $\lambda \in [0, 1]$ and define a new (single) cost $C^{\lambda}_{ij} = \lambda C_{ij} + (1 - \lambda)c_{ij}$ for each edge e_{ij}. Dijkstra's algorithm is then applied to find an optimal path with

respect to the new edge weights C_{ij}^{λ}'s. Any resulting λ-optimal path is Pareto-optimal with respect to our two original criteria. (Indeed, if some other path were **strictly better** based on **both** C_{ij}'s **and** c_{ij}'s, it would also have a lower cumulative cost with respect to C_{ij}^{λ}'s.) The procedure is repeated for different values of λ to obtain more points on $\mathcal{PF}$. The computational complexity of this algorithm is $O(\ell n \log n)$, where ℓ is the number of sampled λ values.

Even though scalarization recovers paths which are rigorously Pareto-optimal and does not rely on any discretization of secondary costs, it has several significant drawbacks. First, the set of λ values to try is a priori unknown. The most straightforward implementation based on a uniform λ-grid on $[0, 1]$ turns out to be highly inefficient since the distribution of corresponding points on the Pareto front is very non-uniform. This is easy to see geometrically, since $(\lambda - 1)/\lambda$ can be interpreted as the slope of the Pareto Front wherever it is smooth; see Figure 4.5A. Indeed, when $\mathcal{PF}$ is not smooth, we typically have infinitely many λ's corresponding to the same point on $\mathcal{PF}$; see Figure 4.5B. To improve the efficiency of scalarization, we have implemented a basic adaptive refinement in λ to obtain the desired resolution of $\mathcal{PF}$. The second drawback is even more significant: when the Pareto front is not convex, a single λ value might define several λ-optimal paths, corresponding to different points on $\mathcal{PF}$; see Figure 4.5(C). In such cases, the Pareto Front typically is not convex, with non-convex portions remaining completely invisible when we use the scalarization. Indeed, this approach can be viewed as approximating $\mathcal{PF}$ as an envelope of "support hyperplanes" and thus only recovers its convex hull [20].

We tested our implementation of this adaptive scalarization algorithm on the example already presented in Figures 4.3 and 4.4. Not surprisingly, our budget-augmented algorithm produces significantly more (approximate) Pareto-optimal solutions than the

scalarization; see Figures 4.4C and 4.4D. Regardless, of the λ-refinement threshold, scalarization completely misses all non-convex portions of $\mathcal{PF}$. In this scenario adapative scalarization produced 35 unique solutions in approximately 1.75 seconds. For comparison, our budget-augmented approach produces 160 (approximate) solutions in just 0.2 seconds.

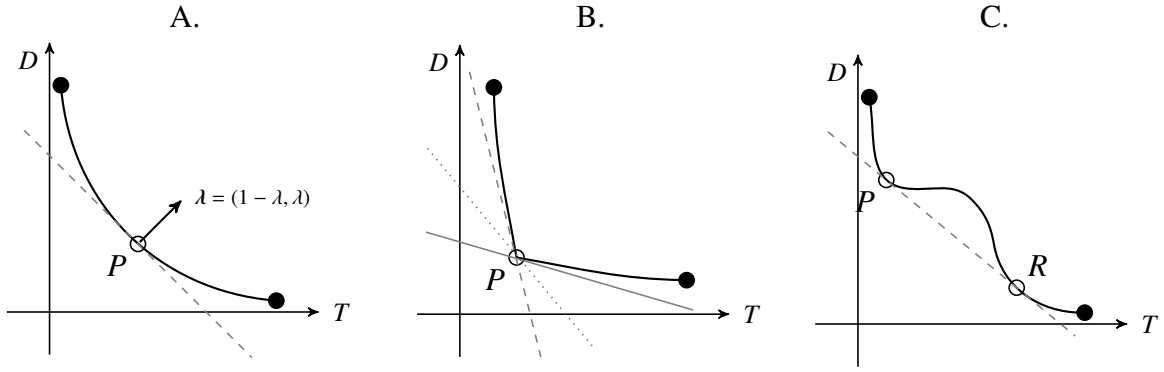


Figure 4.5: (A) Convex smooth Pareto Front with a point P corresponding to some specific $\lambda \in [0, 1]$. The line perpendicular to $\boldsymbol{\lambda}$ is tangent to $\mathcal{PF}$ at P. If any part of $\mathcal{PF}$ fell below it, the path corresponding to P would not be λ-optimal. (B) Convex non-smooth Pareto Front with a 'kink' at the point P makes the corresponding path λ-optimal for a range of λ values, with a different "support hyperplane" corresponding to each of them. (C) Non-convex smooth Pareto Front. Points P and R correspond to 2 different λ-optimal paths. The portion of $\mathcal{PF}$ between P and R cannot be found by scalarization.

4.4.5 Effect of discretization δ

As discussed in Section 4.2, the parameter δ determines the 'discretization' of the secondary budget allowance. For all results we use $\delta = \tilde{V}_j/m$ as defined in Equation (4.7) in Section 4.3.

In Fig. 4.6 we generate a high-density 40,000-node graph (with 3,030,612 edges) and observe the effect of decreasing δ (increasing m) on the results. As δ decreases, it is clear that the produced Pareto Front approaches the true secondary cost curve in gray.

The geometric sequence of m values used to generate the Pareto Fronts in Figs. 4.6A–4.6D is important as discussed in Section 4.3.2 – it is what guarantees the monotone convergence. We note that even for non-geometric sequences of m that non-monotone convergence is difficult to visually observe.

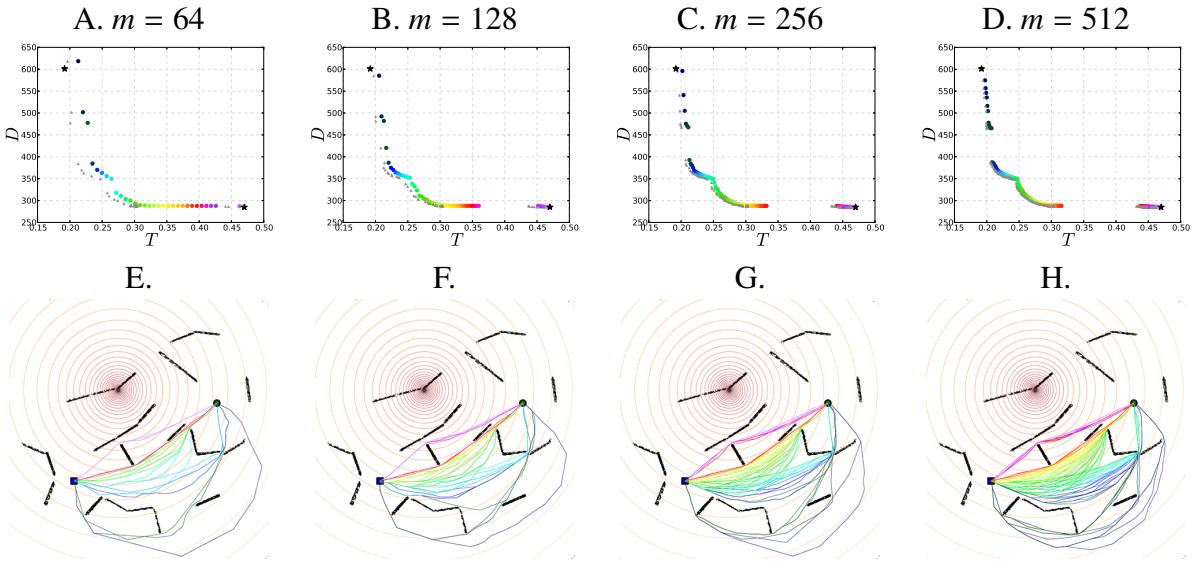


Figure 4.6: Test case with $C = D$ and $c = T$. Results correspond to a $40,000$ node graph with $3,030,612$ edges. (A) – (H): PF and paths as m (number of budget levels) varies. Results are qualitatively the same for $m > 512$.

4.4.6 Swapping primary/secondary costs

Throughout this paper we have assumed that the roles of primary and secondary edge weights are fixed, focused on realistic planning scenarios minimizing the distance traveled subject to constraints on exposure to enemy threat(s). However, an equivalent solution may also be obtained by switching the costs, with a better approximation of $\mathcal{PF}$ sometimes obtained without increasing the number of budget levels [65].

Our experimental evidence indicates that the algorithm does a better job of recovering the portion of the Pareto Front where the slope of the front has smaller magnitude. In

other words, small changes in secondary budget result in small changes in accumulated primary cost, e.g. see Fig. 4.6. Fig. 4.7 shows the result of setting the primary cost to threat exposure ($C = T$) and the secondary cost to be distance ($c = D$). This algorithmic feature can be used to recover a more uniformly dense Pareto Front without significantly increasing the computational cost: consider collating the right portion of Fig. 4.6B and the left part of Fig. 4.7B.

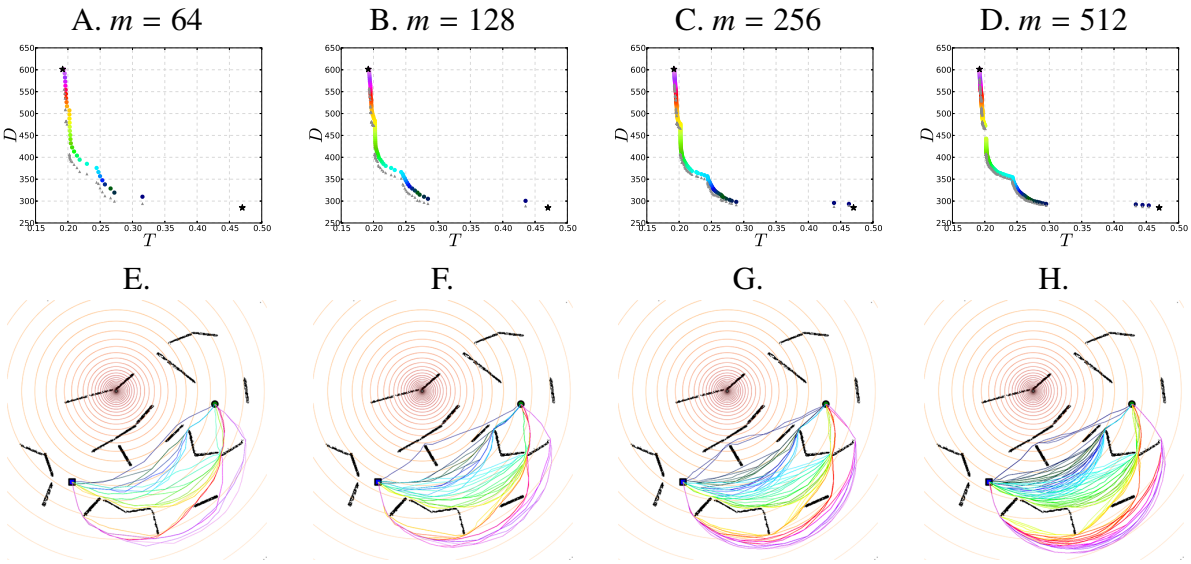


Figure 4.7: Test case with $C = T$ and $c = D$. The same experimental setup as Fig. 4.6 (except with C and c swapped). The Pareto Front is plotted above the "true secondary cost curve" in this figure only (rather than to the right), due to the secondary objective being distance (vertical axis).

4.4.7 Visibility from enemy observer

Previous results focused on an example with a single enemy observer with a simple model for threat exposure. In this final subsection we illustrate the algorithm on a much broader example including multiple enemies with limited visibility behind obstacles. We assume that the size of this domain Ω is 450m $\times$ 450m, and suppose there are two enemy observers with parameters:

Enemy 1	$s_1 = 20$	$\boldsymbol{p}_1 = (225\text{m}, 292.5\text{m})$	$r_1 = 5\text{m}$	$R_1 = +\infty$
Enemy 2	$s_2 = 5$	$\boldsymbol{p}_2 = (225\text{m}, 180\text{m})$	$r_2 = 5\text{m}$	$R_2 = +\infty$

For this problem we take the visibility of enemies into account in the construction of a new threat function T^{vis}. We calculate T^{vis} just as in (4.10), but modify the individual threat equations $\tau_k(\boldsymbol{x})$ to account for limited visibility. We define modified threat level functions

$$\tau_k^{vis}(\boldsymbol{x}) = \begin{cases} \tau_k(\boldsymbol{x}) & \text{if the line segment from } \boldsymbol{x} \text{ to } \boldsymbol{p}_k \text{ is collision-free} \\ \epsilon & \text{otherwise,} \end{cases}$$

for some small $\epsilon > 0$ (we found $\epsilon = 1/\text{Area}(\Omega)$ to work well).

In Fig. 4.8A we show the environment where, just as before, the black rectangles represent obstacles that define the configuration space. The contours of the summed threat exposure functions, $\tau_1^{vis}(\boldsymbol{x}) + \tau_2^{vis}(\boldsymbol{x})$, are plotted. The locations of the two enemies are visually apparent, and the visibility of each enemy can also be easily seen. In particular, the white regions in the image show the points $\boldsymbol{x} \in \Omega$ where visibility is obstructed for both enemies, i.e. $\tau_k^{vis}(\boldsymbol{x}) \equiv \epsilon$ for all $k = 1, 2$. Fig. 4.8B shows the roadmap generated for a $2,048$ node graph with $103,672$ edges in this environment.

Figs. 4.8C & 4.8D show the results with the settings $C = D$ and $c = T^{vis}$. Fig. 4.8C shows the results of the algorithm using the same roadmap (Fig. 4.8B) with $m = 2,048$. The corresponding Pareto Front in Fig. 4.8D exhibits strong non-convexity on this particular domain and generated graph due to the large number of obstacles.

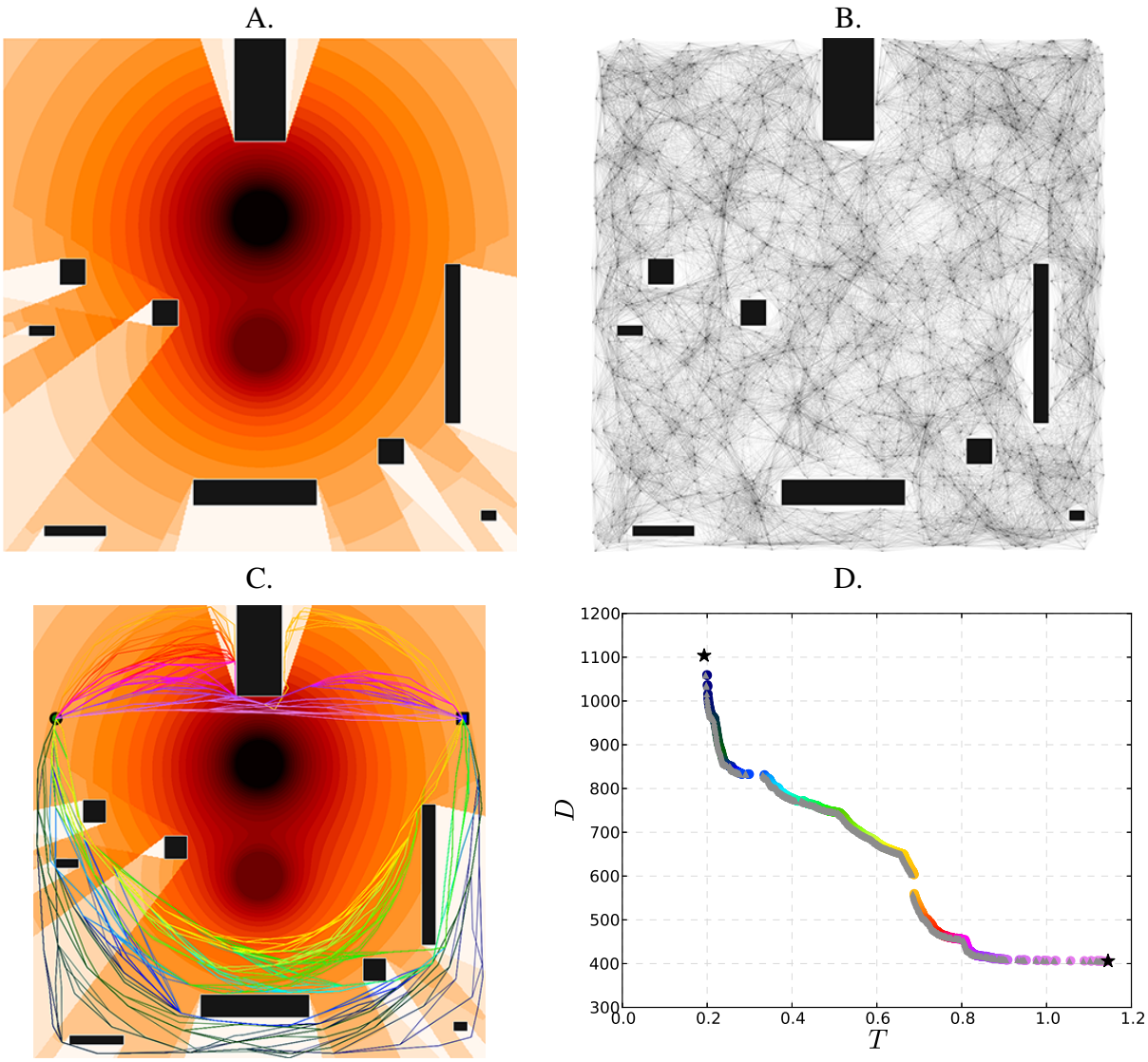


Figure 4.8: Test case with $C = D$ and $c = T^{vis}$. (A) Obstacles plotted with a contour map of the threat exposure (with visibility). (B) Roadmap. (C) & (D) The paths and Pareto Front corresponding to this problem setup, where $m = 2,048$.

Our final test is to use this visibility example to study the convergence of the Pareto Front as the number of nodes in the graph increases. We fix $m = 2048$ budget-levels in order to accurately capture the Pareto Front PF_n corresponding to the generated graph of n nodes. We plot PF_n in Fig. 4.9 as n varies. There is some Pareto Front PF corresponding to a 'continuous' version of this multi-objective problem, and we see that as n

increases PF_n converges to PF.

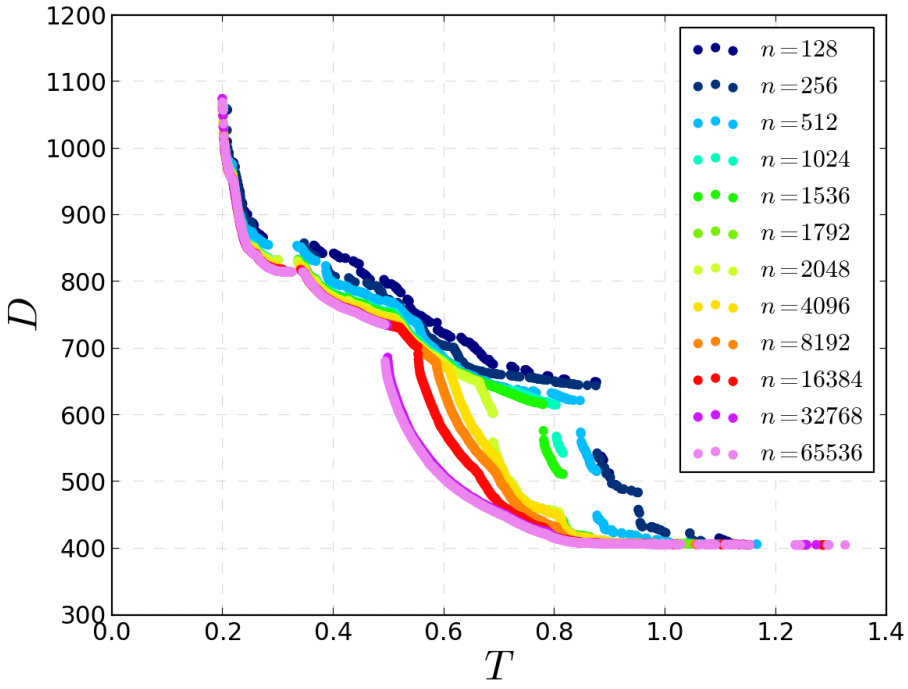


Figure 4.9: Test case with $C = D$ and $c = T^{vis}$. Using the same setup as Fig. 4.8, we study the effect of increasing the number of nodes n in the graph. The budget levels were finely discretized using $m = 2048$.

4.5 Experimental Data

The multi-objective path planning algorithm described in this paper was implemented and demonstrated on a Husky ground vehicle produced by Clearpath Robotics. A picture of the vehicle used in the experiments is shown in Figure 4.10A. The vehicle was equipped with a GPS receiver unit, a WiFi interface, a SICK LMS111 Outdoor 2D LIDAR, an IMU, wheel encoders and a mini-ITX single board computing system with a 2.4GHz Intel i5-520M processor and 8GB of RAM.

Although the GPS unit is used for localization, the associated positional uncertainty

typically cannot meet requirements needed for path planning. Moreover, a map of the environment is required to generate a roadmap to support path planning as discussed in the previous section. To this end, we use a Simultaneous Localization and Mapping (SLAM) algorithm for both localization and mapping, and in particular, the Hector SLAM open source implementation [44]. Hector SLAM is based on scan matching of range data which is suitable for a 2D LIDAR such as the SICK LMS111. The output of Hector SLAM is an occupancy grid representing the environment and a pose estimate of the vehicle in the map. Details of the algorithm and implementation can be found in reference [44]. In order to allow stable and reliable state estimation, we also implemented an extended Kalman filter (EKF). Position estimates were computed by fusing inertial information and SLAM pose estimates through the EKF.

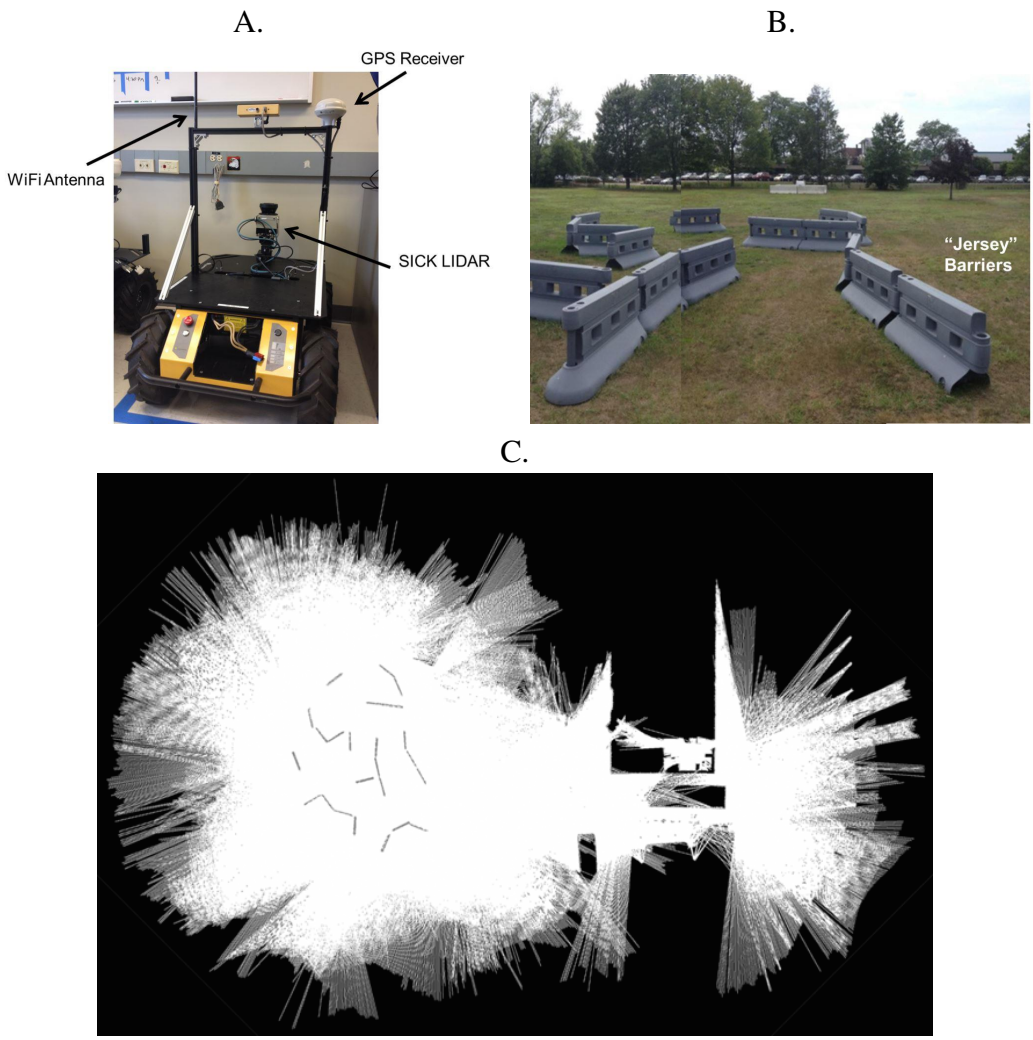


Figure 4.10: (A) Husky robot from ClearPath Robotics. The GPS unit, WiFi antenna and SICK planar LIDAR are shown. A camera is also visible on the roll bar, but is was not used in the experiments. (B) Complex obstacle course created with "Jersey barriers". (C) Map generated by Hector SLAM for the obstacle course.

A set of 24 "Jersey barriers" placed on flat ground were used to create a complex maze-like environment with multiple interconnecting corridors. Figure 4.10B shows the barriers and their positions on the ground. The map generated by Hector SLAM for the obstacle course is shown in Figure 4.10C. The map of the environment is known to the vehicle before the start of the multi-objective planning mission. We utilized the Open Motion Planning Library (OMPL) [77] which contains implementations of numerous

sampling-based motion planning algorithms. We have prior experience using this library for sampling-based planner navigation of an obstacle-rich environment [39]. The generality of OMPL facilitated the development and implementation of our field-capable bi-criteria path planning algorithm: efficient low-level data structures and subroutines are leveraged by OMPL (such as the Boost Graph Library (BGL) [72]) and several predefined robot types (and associated configuration spaces) are available. For the purposes of Husky path planning we apply the PRM algorithm within an $\mathbb{R}^2$ state space.

For the experiments provided in this section, we set the number of budget levels to $m = 768$. The roadmap "grow-time" is set to be about 0.25 seconds, and planning boundary to be a box of size $25m \times 25m$, resulting a roadmap with about $n = 8,000$ vertices and $116,000$ edges (e.g., see Fig. 4.11B). The time to generate the roadmap graph and assign edge-weights is about 1 second, and time to compute the solution and generate the Pareto Front is about 1 second. Thus, a fresh planning instance from graph generation to computing a solution takes about 2 seconds. We also implemented a number of features in the planner intended to add robustness and address contingencies that may arise during the mission, such as when threat values in the environment change. In this case, the execution framework will attempt to re-plan with updated threat values. Since the graph need not be re-generated, replanning is typically much faster (about 1 second).

In the subsequent mission, we assume the Husky is tasked with navigating from its current location to a designated goal waypoint, while avoiding threat exposure to a set of fixed and given threats, i.e., we aim to minimize distance (primary cost) subject to a maximum allowable level of threat exposure (secondary cost). We use the cost functions D (distance) and T (threat exposure) as given in Section 4.2. For threat exposure, we use a single threat with $s = 1$, $r = 0$ and $R = +\infty$. Once the path is computed, the path

is sent down to the lower level path smoothing module. The path smoother is part of the hierarchical planning and execution framework as discussed in [23] and is implemented using model predictive control with realistic vehicle dynamics.

Since threat exposures are difficult to quantify absolutely but easy to compare relatively, we designed a Graphical User Interface (GUI) to pick a suitable path from the Pareto front reconstructed by the proposed algorithm, i.e., the user chooses the shortest path subject to an acceptable amount risk from among the paths in the Pareto front. The process to execute the planning mission is illustrated in Fig. 4.11 can be described as follows:

1. Given an occupancy map of the environment generated by the Hector SLAM algorithm and the known threat location, the user sets the desired goal location for the mission (see Fig. 4.11A). The user has some predefined level of tolerance/budget for the amount of threat exposure allowed. Say, for example, a threat tolerance of 0.79 is permitted in the example shown in Figure 4.11 (for reference, the shortest distance path has a threat exposure of 11).
2. An initial PRM is generated with edge-weights computed by primary and secondary cost functions. Algorithm 6 is executed and the results are used to compute the Pareto Front (see Fig. 4.11B). Next, the path corresponding to each point on the PF is recovered and shown in the GUI, where color indicates the amount of secondary cost (see Fig. 4.11C).
3. From the paths available in Fig. 4.11C and the amount of threat exposure tolerated, an expert analyzes the results. Originally the budget was defined to be 0.79, which corresponds to a path with a length of 32.5m (primary cost). The expert realizes if the exposure allowance is slightly increased to 0.81, a path with length 25.25m is available. These two points correspond to the large vertical drop in the

Pareto Front in Fig. 4.11.

4. From the results and analysis, the user then chooses a path by clicking on a dot in the lower right plot. The picked path is highlighted in bold (see Fig. 4.11C). The user then clicks a button in the GUI to verify the path as desired (see Fig. 4.11D), and finally the selected path is sent to lower level modules for path smoothing and vehicle control.

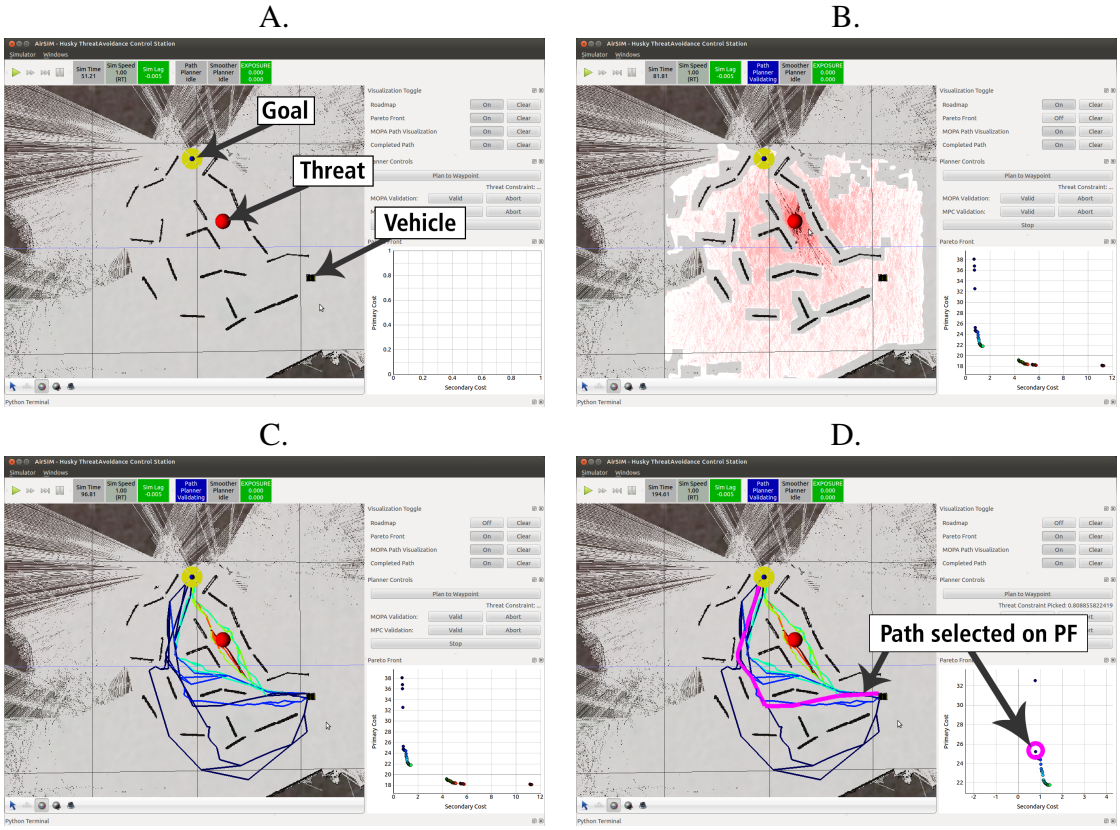


Figure 4.11: (A) The user sets the goal by moving the marker to a location in the occupancy map. The vehicle and threat locations are shown. (B) A roadmap is generated with edges colored according to the secondary cost (threat exposure). The algorithm is run and the Pareto Front is visualized where the vertical and horizontal axes and primary and secondary costs, respectively. (C) The Pareto-optimal paths are shown, each corresponding to a dot on the Pareto Front (by color association). (D) The expert user clicks on a dot in the plot in the lower right corner which highlights the corresponding path in the GUI, choosing a path based on the desired trade-off between primary and secondary cost. The Pareto Front has been zoomed-in for this subfigure.

Fig. 4.12 shows four representative snapshots from a real mission in progress.

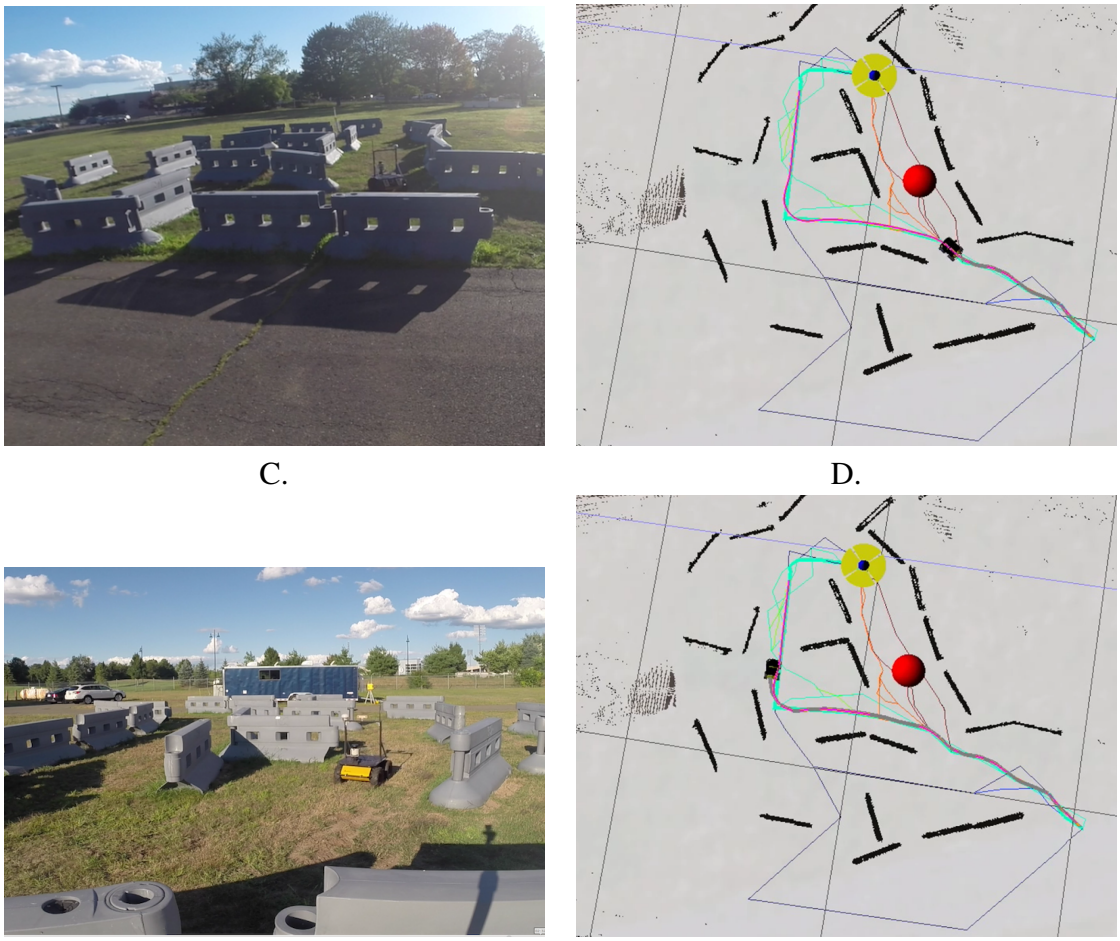


Figure 4.12: At left, images of the experiment; at right, details of the monitoring station showing the status of the vehicle (the black rectangle), the threat location (the red dot), and the optimal trajectories computed by the algorithm. (A) and (B) are the vehicle at near start of the mission and (C) and (D) are the vehicle near the end of the mission.

4.6 Conclusions

We have introduced an efficient and simple algorithm for bi-criteria path planning. Our method has been extensively tested both on synthetic data and in the field, as a component of a real robotic system. Unlike prior methods based on scalarization, our approach

recovers the entire Pareto Front regardless of its convexity. As an additional bonus, choosing a larger value of δ boosts the efficiency of the method, resulting in an approximation of PF, whose accuracy can be further assessed in real time.

The computational cost of our methods is $O(nm)$ corresponding to the number of nodes in the budget-augmented graph. Of course, the total number p of nodes with distinct Pareto optimal cost tuples can be much smaller than nm. In such cases, the prior label-setting algorithms for multi-objective planning will likely be advantageous since their asymptotic cost is typically $O(p \log p)$. However, in this paper we are primarily interested in graphs used to approximate the path planning in continuous domains with edge costs correlated with the geometric distances. As we showed in Section 4.4, for such graphs the number of points on PF is typically quite large (particularly as PRM graphs are further refined), with new Pareto optimal paths becoming feasible on most of the budget levels. Coupled with domain restriction techniques and the simplicity of implementation, this makes our approach much more attractive.

Several extensions would obviously greatly expand the applicability of our method. We hope to extend it to a higher number of simultaneous criteria and introduce A*, D*, and "anytime planning" versions. In addition, it would be very useful to develop a priori bounds for the errors introduced in the PF as a function of δ. Another direction is automating the choice of primary/secondary cost to improve the method's efficiency.

BIBLIOGRAPHY

[1] E. Altman, *Constrained Markov decision processes*, Stochastic Modeling Series, Taylor & Francis, 1999.

[2] K. Alton & I. M. Mitchell, *An Ordered Upwind Method with Precomputed Stencil and Monotone Node Acceptance for Solving Static Hamilton-Jacobi Equations*, Journal of Scientific Computing, Vol. 51, No. 2, pp. 313-348, 2012.

[3] S. Bak, J. McLaughlin, & D. Renzi, *Some Improvements for the Fast Sweeping Method*, SIAM J. Sci. Comput., Vol. 32, No. 5, pp. 2853-2874, 2010.

[4] M. Bardi and I. Capuzzo-Dolcetta, *Optimal Control and Viscosity Solutions of Hamilton Jacobi-Bellman Equations*, Birkhäuser, 1997.

[5] G. Barles and P. E. Souganidis, *Convergence of approximation schemes for fully nonlinear second order equations*, Asymptot. Anal., Vol. 4, pp. 271-283, 1991.

[6] R.E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[7] J. van den Berg, R. Shah, A. Huang, & K. Goldberg, *ANA*: Anytime Nonparametric A**, Association for the Advancement of Artificial Intelligence: Annual Conference (AAAI), San Francisco, CA, 2011.

[8] D.P. Bertsekas, *Dynamic Programming and Optimal Control*, 2nd Edition, Volumes I and II, Athena Scientific, Boston, MA, 2001.

[9] Dimitri P Bertsekas, *A simple and fast label correcting algorithm for shortest paths*, Networks, Vol. 23, No. 8, pp. 703-709, 1993.

[10] Dimitri P Bertsekas, Francesca Guerriero, and Roberto Musmanno, *Parallel asynchronous label-correcting methods for shortest paths*, Journal of Optimization Theory and Applications, Vol. 88, No. 2, pp. 297-320, 1996.

[11] F. Bornemann & C. Rasch, *Finite-element Discretization of Static Hamilton-Jacobi Equations based on a Local Variational Principle*, Computing and Visualization in Science, Vol. 9, No. 2, pp. 57-69, 2006.

[12] M. Boué & P. Dupuis, *Markov chain approximations for deterministic control problems with affine dynamics and quadratic cost in the control*, SIAM J. Numer. Anal., Vol. 36, No. 3, pp.667-695, 1999.

[13] Cacace, S., Cristiani, E., Falcone, M., Picarelli, A. *A patchy Dynamic Programming scheme for a class of Hamilton-Jacobi-Bellman equations*, SIAM J. Sci. Comp. Vol. 34, No. 5, pp. A2625-A2649, 2012.

[14] A. Chacon & A. Vladimirsky, *Fast two-scale methods for Eikonal equations*, SIAM J. on Scientific Computing, Vol. 34, No. 2, 2012.

[15] A. Chacon & A. Vladimirsky, *A parallel Heap-Cell Method for Eikonal equations*, SIAM J. on Scientific Computing, Vol. 37, No. 1, pp. A156-A180, 2015.

[16] A. Chacon, *Eikonal Equations: new two-scale algorithms and error analysis*, Ph.D. Thesis, Cornell University, 2013.

[17] A. Chacon & A. Vladimirsky, *Fast two-scale methods for Eikonal equations*, SIAM J. on Scientific Computing, Vol. 34, No. 2, 2012.

[18] Z. Clawson, A. Chacon, & A. Vladimirsky, *Causal domain restriction for Eikonal equations*, SIAM J. on Scientific Computing, SIAM J. of Scientific Computing, Vol. 36, No. 5, pp. A2478-2505, 2014.

[19] M.G. Crandall, P.-L. Lions, *Viscosity solutions of Hamilton-Jacobi equations*, Transactions of the American Mathematical Society, Vol. 277, No. 1, pp. 1-42, 1983.

[20] I. Das and J.E. Dennis, *A closer look at drawbacks of minimizing weighted sums of objectives for pareto set generation in multicriteria optimization problems*, Struct. Optim., Vol. 14, pp. 63-69, 1997.

[21] R. Dial, *Algorithm 360: Shortest path forest with topological ordering*, Communications of the ACM, Vol. 12, pp. 632-633, 1969.

[22] E.W. Dijkstra, *A Note on Two Problems in Connexion with Graphs*, Numerische Mathematik, Vol. 1, No. 1, pp. 269-271, 1959.

[23] Xuchu Ding, Brendan Englot, Allan Pinto, Alberto Speranzon, and Amit Surana, *Hierarchical multi-objective planning: From mission specifications to contingency management*, Robotics and Automation (ICRA), 2014 IEEE International Conference on, IEEE, pp. 3735-3742, 2014.

[24] Xu Chu Ding, Alessandro Pinto, and Amit Surana, *Strategic planning under uncertainties via constrained Markov decision processes.*, ICRA, IEEE, pp. 4568-4575, 2013.

[25] L.C. Evans, *Partial Differential Equations*, AMS Press, Brooklyn, NY, 1998.

[26] D. Ferguson & A. Stentz, *Field D*: An interpolation-based path planner and replanner*, Proceedings of International Symposium on Robotics Research (ISRR), 2005.

[27] L.R. Ford Jr., *Network Flow Theory*, Paper P-923, Santa Monica, California, RAND Corporation, 1956.

[28] A.V. Goldberg & C. Harrelson, *Computing the Shortest Path: A* Search Meets Graph Theory*, Technical Report, Microsoft Research, 2004.

[29] P.A. Gremaud & C.M. Kuster, *Computational Study of Fast Methods for the Eikonal Equation*, SIAM J. Sci. Comput., Vol. 27, No. 6, pp. 1803-1816, 2006.

[30] E.A. Hansen, S. Zilberstein, & V.A. Danielchenko, *Anytime Heuristic Search: First Results*, Technical Report, 1997.

[31] E.A. Hansen, R. Zhou, *Anytime Hueristic Search*, Journal of Artificial Intelligence Research, Vol. 28, pp. 267-297, 2007.

[32] P.E. Hart, N.J. Nilsson, & B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions of Systems Science and Cybernetics, Vol. SSC-4, No. 2, pp. 100-107, 1968.

[33] S. Hysing & S. Turek, *The Eikonal equation: numerical efficiency vs. algorithmic complexity on quadrilateral grids*, Proceedings of ALGORITMY, pp. 22-31, 2005.

[34] T. Ikeda, & H. Imai *Fast A* Algorithms for Multiple Sequence Alignment*, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.5838`, 1994.

[35] T. Ikeda, & H. Imai *Enhanced A* algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases*, Theoretical Computer Science, Vol. 210, pp. 341-374, 1999.

[36] S. Feyzabadi and S. Carpin, *HMCDP: A hierarchical solution to constrained Markov decision processes*, pp. 3971-3978, May 2015.

[37] J. M. Jaffe, *Algorithms for finding paths with multiple constraints*, Networks, Vol. 14, pp. 95-116, 1984.

[38] O. Junge & H. Osinga, *A set oriented approach to global optimal control*, ESAIM: Control, Optimisation and Calculus of Variations, Vol. 10, No. 2, pp. 259-270, 2004.

[39] S. K. Kannan, W. M. Sisson, D. A. Ginsberg, J. C. Derenick, X. C. Ding, T. A. Frewen, and H. Sane, *Close proximity obstacle avoidance using sampling-based planners*, 2013 AHS Specialists' Meeting on Unmanned Rotorcraft and Network-Centric Operations, 2013.

[40] Sertac Karaman and Emilio Frazzoli, *Sampling-based algorithms for optimal motion planning*, The International Journal of Robotics Research, Vol. 30, No. 7, pp. 846-894, 2011.

[41] Kimmel, R. & Sethian, J.A., *Fast Marching Methods on Triangulated Domains*, Proc. Nat. Acad. Sci., Vol. 95, pp. 8341-8435, 1998.

[42] Lydia E. Kavraki, Petr Svestka, J-C Latombe, and Mark H. Overmars, *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*, IEEE Transactions on Robotics and Automation, Vol. 12, No. 4, pp. 566-580, 1996.

[43] Sven Koenig and Maxim Likhachev, *Fast replanning for navigation in unknown terrain*, Robotics, IEEE Transactions Vol. 21, No. 3, pp. 354-363, 2005.

[44] Stefan Kohlbrecher, Johannes Meyer, Thorsten Graber, Karen Petersen, Uwe Klingauf, and Oskar von Stryk, *Hector open source modules for autonomous mapping and navigation with rescue robots*, RoboCup 2013: Robot World Cup XVII, Springer, pp. 624-631, 2014.

[45] A. Kumar and A. Vladimirsky, *An efficient method for multiobjective optimal control and optimal control subject to integral constraints*, Journal of Computational Mathematics, Vol. 28, pp. 517-551, 2010.

[46] M. Likhachev, G. Gordon, & S. Thrun, *ARA*: Anytime A* with Provable Bounds on Sub-Optimality*, NIPS 16, 2003.

[47] E. Machuca, L. Mandow, and J.L. Pe'rez de la Cruz, *An evaluation of heuristic functions for bicriterion shortest path problems*, New Trends in Artificial Intelligence. Proceedings of the XIV Portuguese Conference on Artificial Intelligence (EPIA), 2009.

[48] Enrique Machuca, Lorenzo Mandow, Jose L. Pérez de la Cruz, and Amparo Ruiz-Sepulveda, *An empirical comparison of some multiobjective graph search algo-*

rithms, Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence (Berlin, Heidelberg), KI'10, Springer-Verlag, pp. 238-245, 2010.

[49] L. Mandow and J. L. Pérez De la Cruz, *A new approach to multiobjective A* search*, Proceedings of the 19th International Joint Conference on Artificial Intelligence (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., pp. 218-223, 2005.

[50] L. Mandow and J. L. Pérez De La Cruz, *Multiobjective A* search with consistent heuristics*, J. ACM, Vol. 57, No. 27, pp. 1-25, 2008.

[51] K. Miettinen, *Nonlinear Multiobjective Optimization*, Kluwer Academic Publishers, International Series in Operations Research & Management Science, Volume 12, 1999.

[52] J.-M. Mirebeau, *Anisotropic Fast-Marching on Cartesian grids using Lattice Basis Reduction*, submitted to SIAM J. on Numerical Analysis; preprint: `http://arxiv.org/abs/1201.1546`

[53] J.-M. Mirebeau, *Efficient fast marching with Finsler metrics*, Numerische Mathematik, Vol. 126, No. 3, pp. 515-557, 2014.

[54] Masahiro Ono, Marco Pavone, Yoshiaki Kuwata, and J. Balaram, *Chance-constrained dynamic programming with application to risk-aware robotic space exploration*, Autonomous Robots, Vol. 39, No. 4, 555-571, 2015.

[55] C. Pêtrès, *Trajectory Planning for Autonomous Underwater Vehicles*, Heriot-Watt University, PhD Dissertation, 2007.

[56] U. Pape, *Implementation and Efficiency of Moore – Algorithms for the Shortest Path Problem*, Math. Programming, Vol. 7, pp. 212-222, 1974.

[57] G. Peyré & L.D. Cohen, *Heuristically Driven Front Propagation for Geodesic Paths Extraction*, Proc. of VLSM '05 (N. Paragios, O. D. Faugeras, T. Chan, C. Schnörr, eds.), Springer, Vol. 3752, pp. 173-185, 2005.

[58] G. Peyré & L.D. Cohen, *Landmark-Based Geodesic Computation for Heuristically Driven Path Planning*, Proc. of CVPR '06, IEEE Computer Society, pp. 2229-2236, 2006

[59] G. Peyré & L.D. Cohen, *Heuristically Driven Front Propogation for Fast Geodesic*

Path Extraction, International Journal for Computational Vision and Biometrics, Vol. 1, No. 1, pp. 55-67, 2008.

[60] R. Philippsen, *A Light Formulation of the E* Interpolated Path Replanner*, Technical report, Autonomous Systems Lab, École Polytechnique Fédérale de Lausanne, 2006.

[61] I. Pohl, *Bi-directional Search*, Machine Intelligence, Vol. 6, eds. Meltzer and Michie, Edinburgh University Press, pp. 127-140, 1971.

[62] I. Pohl, *Heuristic Search Viewed as Path Finding in a Graph*, Artificial Intelligence, Vol. 1, pp. 193-204, 1970.

[63] L. S. Pontryagin, V. Boltyanskii, R. V. Gamkrelidze, & E. F. Mishenko, *The Mathematical Theory of Optimal Processes*, Wiley, 1962.

[64] S. Richter, J.T. Thayer, W. Ruml, *The Joy of Forgetting: Faster Anytime Search via Restarting*, International Conference on Automated Planning and Scheduling, 2010.

[65] Blane Rhoads, Suresh Kannan, Thomas A. Frewen, and Andrzej Banaszuk, *Optimal control of autonomous helicopters in obstacle-rich environments*, United Technologies Research Center, East Hartford, CT (2012).

[66] E. Rouy & A. Tourin, *A Viscosity Solutions Approach to Shape-From-Shading*, SIAM J. Num. Anal., Vol. 29, No. 3, pp. 867-884, 1992.

[67] J.A. Sethian, *A Fast Marching Level Set Method for Monotonically Advancing Fronts*, Proc. Nat. Acad. Sci., Vol. 93, No. 4, pp. 1591-1595, February 1996.

[68] J.A. Sethian, *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision and Materials Sciences*, Cambridge University Press, 1996.

[69] J.A. Sethian & A. Vladimirsky, *Fast Methods for the Eikonal and Related Hamilton–Jacobi Equations on Unstructured Meshes*, Proc. Nat. Acad. Sci., Vol. 97, No. 11, pp. 5699-5703, 2000.

[70] J.A. Sethian & A. Vladimirsky, *Ordered Upwind Methods for Static Hamilton-Jacobi Equations*, Proc. Nat. Acad. Sci., Vol. 98, No. 20, pp. 11069-11074, 2001.

[71] J.A. Sethian & A. Vladimirsky, *Ordered Upwind Methods for Static Hamilton-Jacobi Equations: Theory & Algorithms*, SIAM J. on Numerical Analysis, Vol. 41, No. 1, pp. 325-363, 2003.

[72] J.G. Siek, L.Q. Lee, and A. Lumsdaine, *Boost graph library: User guide and reference manual*, Pearson Education, 2001.

[73] A. J. V. Skriver and K. A. Andersen, *A label correcting approach for solving bi-criterion shortest-path problems*, Computers & Operations Research, Vol. 27, No. 6, pp. 507-524, 2000.

[74] Anthony Stentz, *Optimal and efficient path planning for partially-known environments*, Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on, IEEE, pp. 3310-3317, 1994.

[75] Anthony Stentz, *The focussed D* algorithm for real-time replanning*, IJCAI, Vol. 95, pp. 1652-1659, 1995.

[76] Bradley S. Stewart and Chelsea C. White, III, *Multiobjective A**, J. ACM, Vol. 38, pp. 775-814, 1991.

[77] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki, *The Open Motion Planning Library*, IEEE Robotics & Automation Magazine, Vol. 19, No. 4, pp. 72-82, `http://ompl.kavrakilab.org`, 2012.

[78] Ioan Alexandru Sucan, Mark Moll, and Lydia E. Kavraki, *The open motion planning library*, IEEE Robotics & Automation Magazine, Vol. 19, No. 4, pp. 72-82, 2012.

[79] R. Takei, W. Chen, Z. Clawson, S. Kirov, and A. Vladimirsky, *Optimal control with budget constraints and resets*, SIAM Journal on Control and Optimization, Vol. 53, pp. 712-744, 2015.

[80] J.T. Thayer, *Heuristic search under time and quality bounds*, PhD Thesis, 2012.

[81] J.N. Tsitsiklis *Efficient Algorithms for Globally Optimal Trajectories*, IEEE Tran. Automatic Control, Vol. 40, pp. 1528-1538, 1995.

[82] Chi Tung Tung and Kim Lin Chew, *A multicriteria pareto-optimal path algorithm*, European Journal of Operational Research, Vol. 62, No. 2, pp. 203-209, 1992.

[83] A. Vladimirsky, *Label-setting methods for Multimode Stochastic Shortest Path problems on graphs*, Mathematics of Operations Research, Vol. 33, No. 4, pp. 821-838, 2008.

[84] D.S. Yershov, S.M. LaValle, *Simplicial Dijkstra and A* Algorithms for Optimal Feedback Planning*, in Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2011.

[85] D.S. Yershov, S.M. LaValle, *Simplicial Dijkstra and A* Algorithms: From Graphs to Continuous Spaces*, Advanced Robotics, Vol. 26, No. 17, pp. 2065-2085, 2012.

[86] D.S. Yershov, *Fast Numerical Algorithms for Optimal Robot Motion Planning*, Ph.D. Dissertation, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, Dec. 2013.

[87] F.B. Zhan, F.B., C.E. Noon, *A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths*, Journal of Geographic Information and Decision Analysis, Vol. 4, 2000.

[88] H. Zhao, *A Fast Sweeping Method for Eikonal Equations*, Mathematics of Computation, Vol. 74, No. 250, pp. 603-627, 2004.